

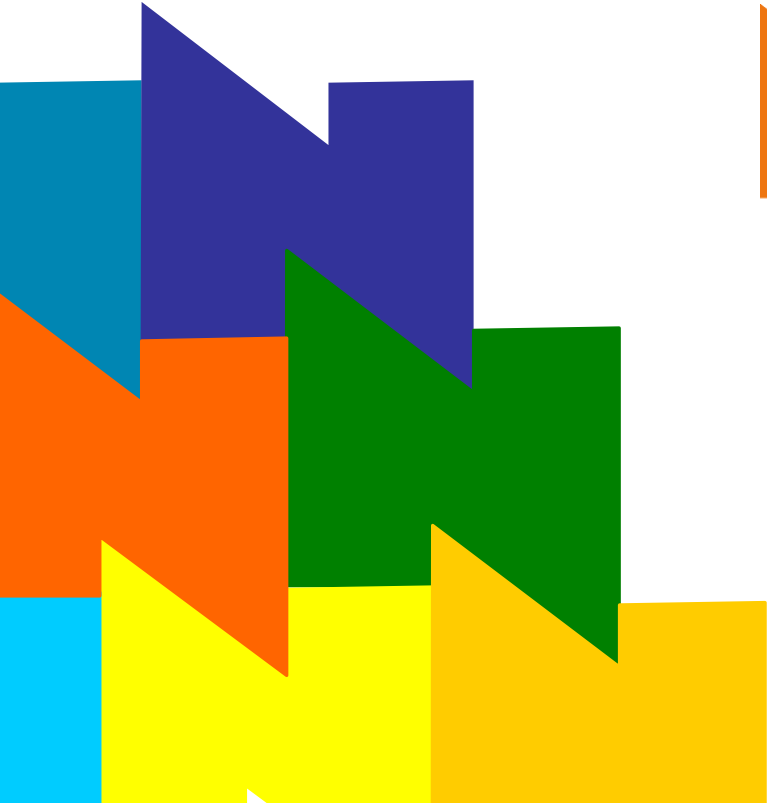


ADI Service Developer's Reference Manual

9000-62162-18



100 Crossing Boulevard
Framingham, MA 01702-5406 USA
www.nmscommunications.com



No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of NMS Communications Corporation.

© 2009 NMS Communications Corporation. All Rights Reserved.

Alliance Generation is a registered trademark of NMS Communications Corporation or its subsidiaries. NMS Communications, Natural MicroSystems, AG, CG, CX, QX, Convergence Generation, Natural Access, Natural Access MX, CT Access, Natural Call Control, Natural Media, NaturalFax, NaturalRecognition, NaturalText, Fusion, Open Telecommunications, Natural Platforms, NMS HearSay, AccessGate, MyCaller, and HMIC are trademarks or service marks of NMS Communications Corporation or its subsidiaries. Multi-Vendor Integration Protocol (MVIP) is a registered trademark of GO-MVIP, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. Windows NT, MS-DOS, MS Word, Windows 2000, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Clarent and Clarent ThroughPacket are trademarks of Clarent Corporation. Sun, Sun Microsystems, Solaris, Netra, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and/or other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Linux is a registered trademark of Linus Torvalds. Red Hat is a registered trademark of Red Hat, Inc. All other marks referenced herein are trademarks or service marks of the respective owner(s) of such marks. All other products used as components within this product are the trademarks, service marks, registered trademarks, or registered service marks of their respective owners.

Every effort has been made to ensure the accuracy of this manual. However, due to the ongoing improvements and revisions to our products, NMS Communications cannot guarantee the accuracy of the printed material after the date of publication or accept responsibility for errors or omissions. Revised manuals and update sheets may be published when deemed necessary by NMS Communications.

P/N 9000-62162-18

Revision history

Revision	Release date	Notes
1.0	May, 2002	MCM, combined 6424-20 and 6341-24 for Natural Access 2002-1
1.1	April, 2003	MCM, Natural Access 2003-1
1.2	December, 2003	SRR, Natural Access 2004-1 Beta
1.3	April, 2004	MCM, Natural Access 2004-1
1.4	November, 2004	SRR, Natural Access 2005-1 Beta
1.5	March 2005	SRR, Natural Access 2005-1
1.6	October, 2005	DEH/LBG, Natural Access 2005-1, SP 1
1.7	July 2006	SRG, Natural Access 2005-1, SP 2
1.8	February, 2009	DEH, Natural Access R8.1
Last modified: January 22, 2009		

Refer to www.nmscommunications.com for product updates and for information about support policies, warranty information, and service offerings.

Table Of Contents

Chapter 1: Introduction	7
Chapter 2: Overview of the ADI service	9
ADI service definition	9
Using ADI with the Natural Call Control service	9
ADI terminology	9
Setting up the Natural Access environment	10
Initializing Natural Access	10
Creating event queues and contexts	10
Opening services	11
Linking with the ADI service	12
Chapter 3: Developing applications.....	13
Recording and playing.....	13
Voice encoding formats	13
Data transfer methods	18
DTMF abort mask	19
Recording	20
Playing.....	25
System restrictions	30
Using simultaneous play and record	30
Performing NMS native play and record	31
NMS native play and record advantages.....	31
Implementing NMS native play and record	32
Native play	33
Native record without inband silence and DTMF detection	35
Native record with inband silence and DTMF detection	38
Managing call progress.....	43
Tone detection	43
Call progress tone events	48
Call progress voice events	48
Call progress termination events.....	49
System restrictions	49
Detecting tones.....	50
Starting tone detection.....	50
Stopping tone detection.....	51
Generating tones.....	52
Playing tones	52
Terminating tone generation	52
System restrictions	52
Collecting digits	53
Synchronous digit collection	53
Asynchronous digit collection.....	55
Modifying DTMF detection	56
Terminating DTMF detection.....	56
Improving DTMF using echo cancellation.....	56
Controlling echo	57
Echo cancellation examples.....	57
Echo canceller components	59
Specifying echo canceller parameters	61

Configuring boards for echo cancellation	61
Recommendations for controlling echo.....	64
Detecting energy.....	66
Starting energy detection	66
Stopping energy detection	67
Detecting voice activity	67
Configuring boards for voice detection	68
Using voice activity detection	68
Sending and receiving FSK data	69
Sending FSK data	69
Terminating FSK data transmission	70
Receiving FSK data	70
Terminating FSK data reception	70
Performing low-level call control	71
Using on-board timers	72
Starting the timer.....	72
Stopping the timer.....	72
Chapter 4: Function summary	73
Telephony protocol functions.....	73
Record and play functions.....	73
Initiating record and play operations	73
Terminating record and play operations	73
Using buffer management functions	73
Using status and modification functions	74
Call progress functions.....	74
Tone detection functions	74
Tone generation functions	75
Digit collection functions.....	75
Echo cancellation functions	75
DTMF and MF detection functions.....	76
Frequency shift key data functions.....	76
Low-level call control functions.....	77
On-board timer functions.....	77
Configuration information functions.....	78
Chapter 5: Function reference	79
Using the function reference	79
adiAssertSignal	80
adiCollectDigits	82
adiCommandEchoCanceller	85
adiCommandRecord.....	91
adiFlushDigitQueue.....	95
adiGetBoardInfo.....	97
adiGetBoardSlots	100
adiGetBoardSlots32	103
adiGetContextInfo	106
adiGetDigit.....	109
adiGetEEPromData	111
adiGetEncodingInfo	113
adiGetPlayStatus.....	115
adiGetRecordStatus	117
adiGetTimeStamp.....	119

adiInsertDigit.....	121
adiModifyEchoCanceller.....	123
adiModifyPlayGain	128
adiModifyPlaySpeed	130
adiPeekDigit	131
adiPlayAsync	132
adiPlayFromMemory	136
adiQuerySignalState	139
adiRecordAsync.....	141
adiRecordToMemory	145
adiSetBoardClock	149
adiSetNativeInfo	150
adiStartCallProgress	154
adiStartDial	158
adiStartDTMF.....	161
adiStartDTMFDetector.....	163
adiStartEnergyDetector.....	165
adiStartMFDetector.....	167
adiStartPlaying	170
adiStartProtocol	174
adiStartPulse	177
adiStartReceivingFSK.....	179
adiStartRecording.....	182
adiStartSendingFSK.....	186
adiStartSignalDetector	189
adiStartTimer	192
adiStartToneDetector.....	194
adiStartTones	198
adiStopCallProgress.....	201
adiStopCollection	203
adiStopDial.....	204
adiStopDTMFDetector	206
adiStopEnergyDetector	208
adiStopMFDetector	210
adiStopPlaying	212
adiStopProtocol.....	213
adiStopReceivingFSK	215
adiStopRecording	216
adiStopSendingFSK	217
adiStopSignalDetector.....	218
adiStopTimer	220
adiStopToneDetector	222
adiStopTones.....	224
adiSubmitPlayBuffer	226
adiSubmitRecordBuffer	228
Chapter 6: Demonstration programs.....	231
Summary of the demonstration programs	231
ctademo.c and ctademo.h.....	231
Host port to port connection: hostp2p	232
Play and record: playrec.....	234
Multi-threaded application: threads.....	236

Chapter 7: Errors	237
Alphabetical error summary	237
Numerical error summary	239
Chapter 8: Events	241
Event data structure	241
DONE events	242
Alphabetical event summary	243
Numerical event summary	245
Events ordered by category	247
Administrative events.....	247
Play and record events	248
DTMF events.....	248
MF events	248
Call progress events	249
Tone detector events	250
Call control primitives	250
Miscellaneous events.....	251
Chapter 9: Parameters	253
Overview of the ADI service parameters	253
ADI_CALLPROG_PARMS	254
ADI_COLLECT_PARMS.....	256
ADI_DIAL_PARMS	257
ADI_DTMF_PARMS	259
ADI_DTMFDETECT_PARMS	260
ADI_ENERGY_PARMS.....	260
ADI_FSKRECEIVE_PARMS.....	261
ADI_FSKSEND_PARMS	261
ADI_PLAY_PARMS	261
ADI_RECORD_PARMS	262
ADI_START_PARMS	264
ADI_TONE_PARMS	267
ADI_TONEDETECT_PARMS.....	268
Chapter 10: DSP files.....	269
DSP file summary.....	269

1

Introduction

The *ADI Service Developer's Reference Manual* describes how to develop an application using the ADI service in the Natural Access environment. It also provides detailed descriptions of the ADI functions. Use this reference manual with the *Natural Access Developer's Reference Manual*.

This document is intended for developers of telephony and voice applications who are using Natural Access. This document defines telephony terms where applicable, but assumes that you are familiar with telephony concepts and the C programming language.

2

Overview of the ADI service

ADI service definition

The ADI service is a C function library component of Natural Access that enables application programs to execute multiple telephony functions on NMS Communications AG boards, CG boards, QX boards, and PacketMedia HMP software.

The ADI service provides the following functionality:

- Play/record
- Tone detection
- Tone generation
- DTMF collection
- Echo cancellation
- Auxiliary functions such as energy detection, voice activity detection, FSK data transmission and reception, low-level call control, on-board timers, and board functions
- Call progress

Using ADI with the Natural Call Control service

The Natural Call Control (NCC) service is the standard call control for NMS Communications products. The NCC service replaces all call control functions that the ADI service formerly provided.

For more information about the NCC service, refer to the *Natural Call Control Service Developer's Reference Manual*.

ADI terminology

A port is the object on which telephony functions are performed. It contains physical and logical resources on the board. A port is represented by a context and a context handle (**ctahd**), a software handle that enables the application (and its developer) to keep track of software activities.

To access most functionality on a port, the application must associate a telephony protocol with the port. On AG and CG hardware, the telephony protocol is embodied by a trunk control program (TCP), and must be loaded during board initialization. NMS Communications provides TCPs for most standard telephone line interfaces. Starting a protocol enables the use of call control functions. Almost all functions require a protocol to be loaded. For applications that do not use call control functions or choose to manage the line interface manually, the NOCC (no call control) TCP is provided.

For more information about controlling calls under specific TCPs, refer to the *NMS CAS for Natural Call Control Developer's Reference Manual*.

Setting up the Natural Access environment

Before you can call functions from the ADI library, the application must initialize Natural Access and open the ADI service. Application setup for Natural Access consists of the following steps:

1. Initialize Natural Access for the process.
2. Create event queues and contexts.
3. Open services on each context.

To set up a second Natural Access application that shares a context with the first application:

1. Initialize the Natural Access application.
2. Create event queues.
3. Attach the application to the existing context.

Initializing Natural Access

Initialize Natural Access by calling **ctaInitialize** and specifying the service and service manager names. Service managers are dynamic link libraries (DLLs) in Windows and shared libraries in UNIX. Only the services initialized in the call to **ctaInitialize** can be opened by the application.

Use one of the following service managers in your call to **ctaInitialize**:

Board family	Manager
AG	ADIMGR
CG	ADIMGR
QX	QDIMGR

Note: The PacketMedia HMP process is a member of the CG board family.

Creating event queues and contexts

After initializing Natural Access, create the event queues and contexts.

Create one or more event queues by calling **ctaCreateQueue** and specifying the service managers to attach to each queue. The ADI service manager is ADIMGR (or QDIMGR). When you attach a service manager to a queue, you make that service manager available to the queue.

Create a context by calling **ctaCreateContext** and providing the queue handle (**ctaqueuehd**) that was returned from **ctaCreateQueue**. All events for services on the context are received in the specified event queue. **ctaCreateContext** returns a context handle (**ctahd**), which the application supplies when invoking ADI service functions. Events communicated back to the application are also associated with the context.

Refer to the *Natural Access Developer's Reference Manual* for details on the programming models created by the use of contexts and event queues.

Opening services

Open services on a context by calling **ctaOpenServices**. When opening the ADI service, specify a context, a specific board, a timeslot, and a mode. The parameter structure CTA_MVIP_ADDR contains the following fields: board, bus, stream, timeslot, and mode. For all boards, bus and stream can be 0.

The board field specifies the board number you want to use. For AG and CG boards, refer to the system configuration file for the board keyword identifying each board in the system. See the *NMS OAM System User's Manual* for more information. For QX boards, refer to the *qx.cfg* configuration file for board identification. See the *QX 2000 Installation and Developer's Manual* for more information.

The timeslot and mode fields are used to calculate which timeslots to allocate to the service. The timeslot specifies the base timeslot, and the mode dictates how many timeslots are allocated.

The mode field can be one of the following values:

Value	Description
ADI_VOICE_INPUT	Receives in-band data only. The data is received by the DSP on the given timeslot.
ADI_VOICE_OUTPUT	Transmits in-band data only. The data is transmitted by the DSP on the given timeslot.
ADI_VOICE_DUPLEX	Receives and transmits in-band data on the given timeslot. Typically used with the NOCC protocol and allows media (for example, voice and fax) reception and transmission.
ADI_FULL_DUPLEX	This is both ADI_SIGNAL_DUPLEX and ADI_VOICE_DUPLEX. The port receives and transmits both in-band media and out-of-band signaling.

Use the demonstration program *ctatest* to verify that the ADI service is properly installed. Refer to the *Natural Access Developer's Reference Manual* for more information about *ctatest*.

When you open the ADI service, specify a DSP address. A DSP address is specified as a timeslot. Bus and stream fields are 0 (zero). The following table shows valid timeslot values for NMS Communications boards:

Board	Timeslot
CG 6000/C	0 - 127
CG 6100C	0 - 599
CG 6500C, CG 6565/C, CG 6060/C	0 - 511
PacketMedia HMP	0 - maximum determined by license agreement
AG 4000/C, AG 4040/C	0 - 127
AG 2000, AG 2000-BRI	0 - 7
AG 2000C	0 - 23
QX 2000	0 - 3

When the ADI service manager is attached to an event queue, it opens the board driver and associates the muxable wait object returned by the driver open command with the event queue. When this wait object is signaled on receipt of events from the board, **ctaWaitEvent** processes the events through the ADI service and passes any event generated back to the calling function.

Using the ADI service in driver-only mode

To access only the board driver, use the special board argument `ADI_AG_DRIVER_ONLY` in place of a real board number in the `CTA_MVIP_ADDR` structure. This argument permits the application to use a virtual port. The application can use the following functions on a context in driver-only mode, since they do not require physical board resources:

- **adiGetBoardInfo**
- **adiGetBoardSlots**
- **adiGetBoardSlots32**
- **adiGetTimeStamp**
- **adiGetEEPromData**

adiGetTimeStamp and **adiGetEEPromData** are not available for QX boards.

Note: All other functions that take a context handle (**ctahd**) require board-level resources.

Linking with the ADI service

The ADI service contains two components, the ADI service interface and the ADI service implementation. When building a Natural Access application that uses the ADI service, link to *adiapi.lib* (under UNIX, *libadiapi.so*).

For existing applications, modify the make files to link with *adiapi.lib* (*libadiapi.so*). Since earlier applications using the ADI service linked to *adimgr.lib* (under UNIX, *libadimgr.so*), it is included only for backward compatibility.

See the *Natural Access Service Writer's Manual* for more details about service implementation.

3

Developing applications

Recording and playing

The most convenient way to program playing and recording applications is to use the Voice Message service since it provides disk management with the playing and recording functionality. The Voice Message service uses the ADI service device level record and play functions. To use the Voice Message service with the ADI service playing and recording functions, open both services on the same context.

When using the Voice Message service, you do not call the ADI playing and recording functions directly. The Voice Message service calls the functions when needed. For more information, refer to the *Voice Message Service Developer's Reference Manual*.

To create an application using your own disk management functions, call the ADI functions directly.

This topic presents:

- Voice encoding formats
- Data transfer methods
- DTMFabort mask
- Recording
- Playing
- System restrictions
- Using simultaneous play and record

Voice encoding formats

When recording or playing speech files, you must select an encoding format. The primary issue to consider when selecting a format is the compression ratio and the fidelity. More aggressive compression requires less disk space and reduces host-to-board loading, but uses more DSP resources.

Each encoding format has a minimum data block size, called a frame. Frames vary in size and duration depending upon the encoding format. For NMS Communications boards, a frame corresponds to 10 or 20 milliseconds of speech, depending on the encoding format.

AG and CG boards support the following encoding formats:

- NMS Communications ADPCM 16, 24, 32 kbit/s and PCM framed 64 kbit/s
- G.726-compliant ADPCM
- G.723.1 5.3 kbit/s and 6.3 kbit/s (CG boards only)
- G.729A 8 kbit/s (CG boards only)
- OKI ADPCM (24 and 32 kbit/s)
- A-law and mu-law PCM
- 8-bit and 16-bit linear PCM at 11 kilo-samples per second
- 16-bit linear PCM at 8 kilo-samples per second
- IMA ADPCM (24 and 32 kbit/s)
- Microsoft GSM full rate

QX boards support the following encoding formats:

- NMS Communications ADPCM 16, 24, 32 kbit/s and PCM framed 64 kbit/s
- G.726 16, 24, 32, and 40 kbit/s
- OKI ADPCM (24 and 32 kbit/s)
- A-law and mu-law PCM
- 8-bit and 16-bit linear PCM at 11 kilo-samples per second
- 16-bit linear PCM at 8 kilo-samples per second
- IMA ADPCM (24 and 32 kbit/s)
- VOX ADPCM 32 kbit/s

The PacketMedia HMP process supports the following encoding formats:

- NMS Communications ADPCM 16, 24, 32 kbit/s and PCM framed 64 kbit/s
- G.726-compliant ADPCM 32 kbit/s
- OKI ADPCM (32 kbit/s)
- IMA ADPCM (32 kbit/s)
- A-law and mu-law PCM
- 16-bit linear PCM at 8 kilo-samples per second

The encodings refer to the data going to and from the host, typically stored in a voice file. With the exception of ADI_ENCODE_NMS_64, host encoding is independent of line encoding, which is always either mu-law or A-law depending on how the board is configured when it is initialized.

The following table lists the ADI encoding formats:

Encoding format	Description	Sample size (bits)	Sample rate (Hz)	Frame size (bytes)	Frame time (ms)	Data rate (bytes/sec)
ADI_ENCODE_NMS_16	NMS Communications ADPCM 16 kbit/s	2	8000	42	20	2100
ADI_ENCODE_NMS_24	NMS Communications ADPCM 24 kbit/s	3	8000	62	20	3100
ADI_ENCODE_NMS_32	NMS Communications ADPCM 32 kbit/s	4	8000	82	20	4100
ADI_ENCODE_NMS_64	Framed PCM 64 kbit/s	8	8000	162	20	8100
ADI_ENCODE_MULAW	mu-law 64 kbit/s	8	8000	80	10	8000
ADI_ENCODE_ALAW	A-law 64 kbit/s	8	8000	80	10	8000
ADI_ENCODE_EDTX_MULAW	mu-law 64 kbit/s with EDTX headers	8	8000	82	10	8000
ADI_ENCODE_EDTX_ALAW	A-law 64 kbit/s with EDTX headers	8	8000	82	10	8000
ADI_ENCODE_PCM8M16	PCM 8 kss 16 bit mono (WAVE)	16	8000	160	10	16000
ADI_ENCODE_OKI_24	OKI ADPCM 24 kbit/s	4	6000	30	10	3000
ADI_ENCODE_OKI_32	OKI ADPCM 32 kbit/s	4	8000	40	10	4000
ADI_ENCODE_PCM11M8	PCM 11 kss 8 bit mono (WAVE)	8	11000	110	10	11000
ADI_ENCODE_PCM11M16	PCM 11 kss 16 bit mono (WAVE)	16	11000	220	10	22000
ADI_ENCODE_G723_5	ITU G.723.1 5.3 kbit/s	N/A	8000	20	30	667
ADI_ENCODE_G723_6	ITU G.723.1 6.3 kbit/s	N/A	8000	24	30	800
ADI_ENCODE_EDTX_G723_5	ITU G.723.1 5.3 kbit/s with EDTX headers	N/A	8000	22	30	667

Encoding format	Description	Sample size (bits)	Sample rate (Hz)	Frame size (bytes)	Frame time (ms)	Data rate (bytes/sec)
ADI_ENCODE_EDTX_G723_6	ITU G.723.1 6.3 kbit/s with EDTX headers	N/A	8000	26	30	800
ADI_ENCODE_EDTX_G723	ITU G.723.1 with EDTX headers	N/A	8000	26	30	800
ADI_ENCODE_G726	ITU G.726 ADPCM 32 kbit/s	4	8000	40	10	4000
ADI_ENCODE_EDTX_G726	ITU G.726 ADPCM 32 kbit/s with EDTX headers	4	8000	42	10	4000
ADI_ENCODE_G726_16	ITU G.726 ADPCM 16 kbit/s	2	8000	Variable	Variable	2000
ADI_ENCODE_G726_24	ITU G.726 ADPCM 24 kbit/s	3	8000	Variable	Variable	3000
ADI_ENCODE_G726_32	ITU G.726 ADPCM 32 kbit/s	4	8000	Variable	Variable	4000
ADI_ENCODE_G726_40	ITU G.726 ADPCM 40 kbit/s	5	8000	Variable	Variable	5000
ADI_ENCODE_G729A	ITU G.729A 8 kbit/s	N/A	8000	10	10	1000
ADI_ENCODE_EDTX_G729A	ITU G.729A 8 kbit/s with EDTX headers	N/A	8000	12	10	1000
ADI_ENCODE_IMA_24	IMA ADPCM 24 kbit/s	4	6000	36	10	3600
ADI_ENCODE_IMA_32	IMA ADPCM 32 kbit/s	4	8000	46	10	4600
ADI_ENCODE_VOX_32	VOX ADPCM 32 kbit/s	4	8000	40	10	4000
ADI_ENCODE_GSM	MS-GSM 13 kbit/s	N/A	8000	130	80	1625

Note: The Voice Message service has equivalent encoding formats with names that begin with VCE_.

DSP files

When recording or playing speech files on AG boards, a specific DSP file must be loaded for each encoding type.

When recording or playing speech files on CG boards, a specific DSP file must be loaded for each encoding type except when using the native play and record feature. The native play and record feature combines an ADI port with an MSPP endpoint and plays or records speech data directly to or from an IP endpoint with no transcoding. Native play and record supports:

- Playing media files recorded from streams that contain SID frames, silence, RFC 2833 frames, and lost frame markers.
- Silence and DTMF detection and reporting while recording RTP streams.

For information on the native play and record feature, refer to *Performing NMS native play and record* on page 31.

The previous table lists the ADI_ENCODE_EDTX encoding formats to use for native recording. For native playing, use either the ADI_ENCODE_EDTX or ADI_ENCODE encoding formats. **adiSetNativeInfo** sets play and record parameters.

For QX boards, the standard DSP file supports the valid encoding types. For more information, see *Encoding formats and DSP files* on page 134. The table lists the DSP files that must be loaded on the AG and CG boards. It also lists the valid encoding types that QX boards and PacketMedia HMP processes support.

Buffer sizes

Except for buffers that contain speech data recorded in one of the ADI_ENCODE_EDTX encoding formats, all buffers submitted to the ADI service play functions must be large enough to contain an integral number of frames for the selected encoding format. For example, if you select ADI_ENCODE_NMS_24, the buffer size must be a multiple of 62 bytes. Failure to submit a buffer meeting this size requirement causes the play function to terminate with CTAERR_BAD_SIZE. For ADI_ENCODE formatted data without EDTX headers that meet the multiple frame size requirement, buffers submitted to the ADI service can be any size.

Use the ADI_ENCODE_EDTX encoding formats to record speech data directly from an IP endpoint. Buffers recorded from encoded RTP codec streams can contain variable size frames and must contain marker frames representing silence and discontinuous transmission (DTX) periods. These characteristics do not guarantee that any given buffer size will contain an integer multiple of codec frames, marker frames, or both. Therefore, buffers containing ADI_ENCODE_EDTX formatted data submitted to the ADI service can be any size.

Each board has a physical buffer size that is both board and encoding dependent. If you submit a buffer larger than the physical size, the ADI service divides the buffer into physical segments and submits those segments to the board. To eliminate fractional buffers and to reduce the board-to-host interactions, the optimum user buffer will be a multiple of the physical buffer size. This size is retrieved with **adiGetEncodingInfo**.

The ADI service employs a double-buffering scheme when recording and playing voice files. When the board finishes processing a buffer, the application must already have allocated and submitted the subsequent buffer to the ADI service.

On heavily loaded systems, the throughput requirements between the host and the board can cause gaps in the voice record or playback. This is called an underrun condition. Failure to maintain pace with the board can also cause underruns in the voice record or playback. Greater file compression may be necessary to eliminate the problem.

The ADI service counts the number of underruns that occur, but not the duration. Call **adiGetRecordStatus** and **adiGetPlayStatus** to retrieve the underrun count.

Note: Do not submit small buffers (buffers that hold less than one second of data). Small buffers can also cause underruns. Derive the data throughput for a given encoding method from the **adiGetEncodingInfo** return values.

Data transfer methods

The ADI service provides three methods by which the application can transfer speech data to and from the board:

Method	Description
Single memory transaction	The application submits a single data buffer to the ADI service.
Asynchronous transfer	The application serially submits multiple buffers by exchanging commands and events with the ADI service.
Callback transfer	The ADI service manages the buffers and invokes an application callback function to retrieve or store data.

The functions used to initiate play or record depend upon the data transfer method selected, as shown in the following table:

Operation	Single memory	Asynchronous	Callback
Play	adiPlayFromMemory	adiPlayAsync	adiStartPlaying
Record	adiRecordToMemory	adiRecordAsync	adiStartRecording

adiStartPlaying and **adiStartRecording** are not supported when Natural Access is running in client/server mode. For more information, refer to the *Natural Access Developer's Reference Manual*.

Single memory transaction

If the application invokes **adiPlayFromMemory** or **adiRecordToMemory**, it supplies a single buffer that is retained by the ADI service for the duration of the function. The ADI service divides the application buffer into physical segments and performs all handshaking with the board.

Note: A buffer submitted for playing can be shared by multiple instances of the play function (within the same process) but the buffer submitted for recording must be unique for each active recording instance.

When the ADI service delivers ADIEVN_PLAY_DONE or ADIEVN_RECORD_DONE to the application, the buffer is then available for reuse or disposal.

In summary:

- Single memory transaction is relatively simple and minimizes application interaction.
- Single memory transaction consumes a large virtual address space for large voice files. An application may experience latency reading or writing large files to and from storage.

Asynchronous transfer

The asynchronous transfer method gives you maximum latitude with buffer address, size, and submission. When the play or record function is started with **adiPlayAsync** or **adiRecordAsync**, an initial buffer is submitted. Whenever the board starts a new buffer, an event is generated. The application must submit a new buffer (using **adiSubmitPlayBuffer** or **adiSubmitRecordBuffer**) before the board finishes the current buffer.

In summary:

- The programmer can control buffer addresses and sizes with asynchronous transfer, and have asynchronous access to a storage medium.
- Asynchronous transfer is more complicated to program.

Callback transfer

The callback transfer method balances simplicity in programming and resource consumption. The ADI service allocates the buffers and invokes an application-specified callback function whenever a buffer needs to be filled (during a play function) or when a buffer needs to be emptied (during a record function). Within the callback routine, the application synchronously accesses the storage medium before returning.

In summary:

- Callback transfer minimizes virtual memory consumption and is simple to implement.
- The application cannot control the buffer addresses or sizes with callback transfer, and it requires synchronous access to a storage medium.

DTMFabort mask

By default, the board terminates play and record when any DTMF key is entered. You can specify which DTMF keys terminate the function using the DTMFabort mask in *ADI_PLAY_PARMs* on page 261 or *ADI_RECORD_PARMs* on page 262.

The DTMFabort mask is a 16-bit entity in which each bit corresponds to a specific key on the telephone keypad. Setting a bit in the mask terminates the voice function if that particular key is entered. The DTMFabort mask corresponds to the DTMF telephone keys as shown:

Most significant bit to least significant bit																
Bit position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DTMF key	D	C	B	A	#	*	9	8	7	6	5	4	3	2	1	0

For example, if the abort mask is set to 0x03FF, the play or record function terminates if the remote party enters any digit from 0 through 9. The *adidef.h* include file contains `#defines` (`ADI_DTMF_xxx`) for each digit and for certain digit groups.

Note: The `DTMFabort` mask has no effect on digit collection.

If any digits are queued in the ADI service when a play or record voice operation is started, and the voice operation is to terminate on those specific touchtones, the voice operation terminates immediately. To prevent this from happening, use **adiFlushDigitQueue** or **adiGetDigit** to remove the escape key from the queue.

The digit queue is automatically flushed when a call is released.

Recording

The `ADI_RECORD_PARMS` structure contains the record function parameters.

Initiating record

The ADI service provides three functions to initiate voice record. The function used depends upon the data transfer method.

Use this function...	When...
adiRecordToMemory	The application submits a single buffer to the ADI service.
adiStartRecording	The ADI service invokes an application-specified callback function when a buffer is full. The application must store the data before returning. Note: Applications running in client/server mode do not support adiStartRecording .
adiRecordAsync	The ADI service generates a buffer full event when each buffer is full. The application asynchronously stores the data and submits empty buffers in response.

The ADI service returns `SUCCESS` if the recording function successfully started.

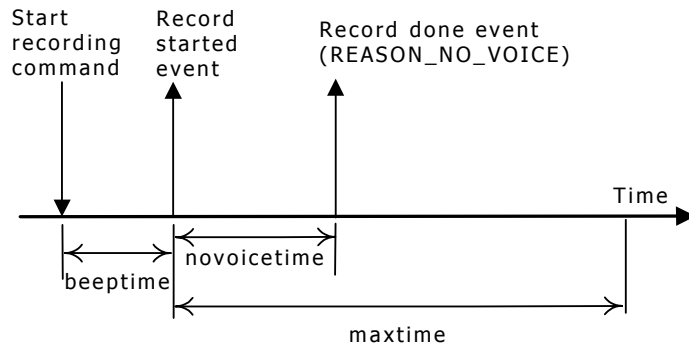
Terminating record

The record function terminates when the ADI service delivers `ADIEVN_RECORD_DONE`, regardless of the transfer method. The event value field contains one of the following termination reasons:

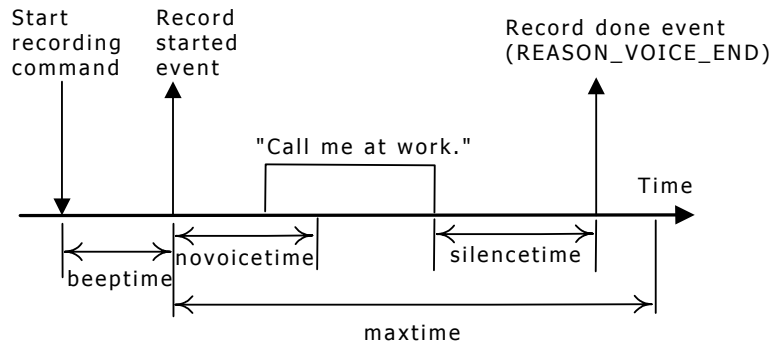
If...	Then play ends with...
The call was released by either party	<code>CTA_REASON_RELEASED</code>
A DTMF digit specified in the abort mask was entered by the remote party	<code>CTA_REASON_DIGIT</code>
The application aborted recording with adiStopRecording	<code>CTA_REASON_STOPPED</code>
The remote party never spoke (see the no voice illustration)	<code>CTA_REASON_NO_VOICE</code>
The remote party stopped speaking for the voice end time period (see the voice end illustration)	<code>CTA_REASON_VOICE_END</code>
The remote party spoke longer than the maximum duration (see the timeout illustration)	<code>CTA_REASON_TIMEOUT</code>

Record termination - no voice

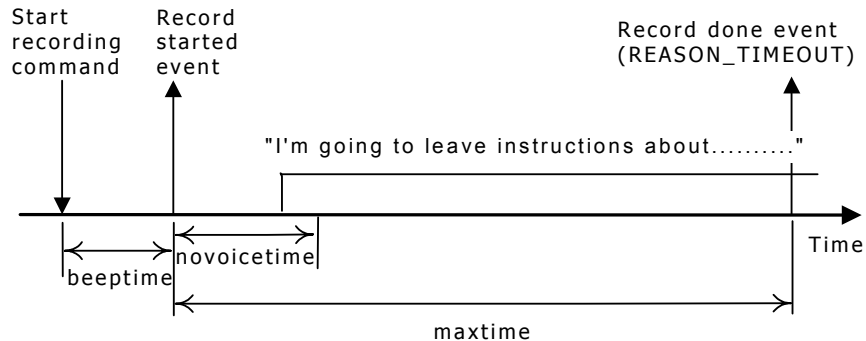
The following illustration shows record termination - no voice:

**Record termination - voice end**

The following illustration shows record termination - voice end:

**Record termination - timeout**

The following illustration shows record termination - timeout:



Three timer parameters terminate the record function:

Parameter	Description
novoicetime	Time, in milliseconds, that the remote party has after the beep-sync prompt to start speaking. novoicetime is stored in the ADI_RECORD_PARMS structure.
silencetime	Maximum silence duration, in milliseconds, after the remote caller has stopped speaking. silencetime is stored in the ADI_RECORD_PARMS structure.
maxtime	Record function time limit, in milliseconds. The remote caller has maxtime milliseconds after the beep to completely record a message. maxtime is a function argument specified when initiating the record function.

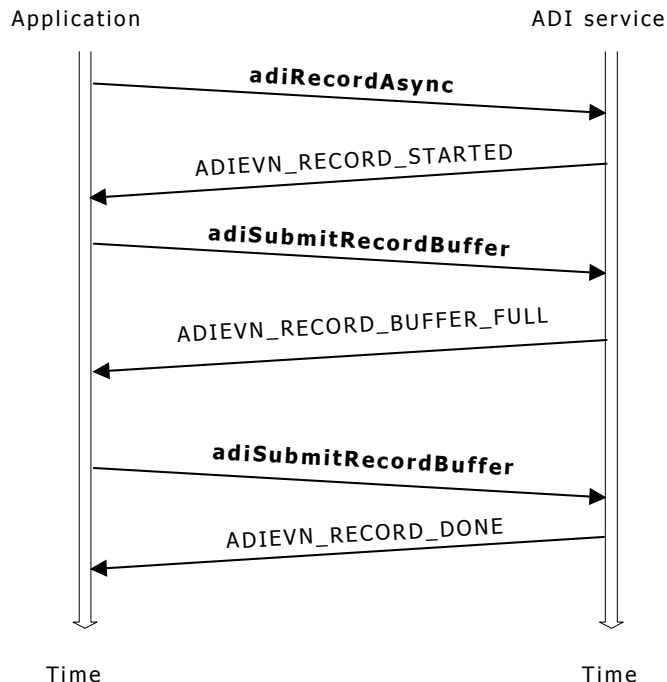
Data transfer using callback mode

In record callback mode, the ADI service allocates two record buffers when the record function initiates. The ADI service invokes the application-specified callback routine whenever a record buffer is filled. You specify the callback function when you initiate record with **adiStartRecording**.

When the ADI service fills a record buffer, it invokes the record callback function and passes it the buffer pointer and the buffer size. The callback routine writes the data to a storage medium such as a disk and returns.

Data transfer using asynchronous mode

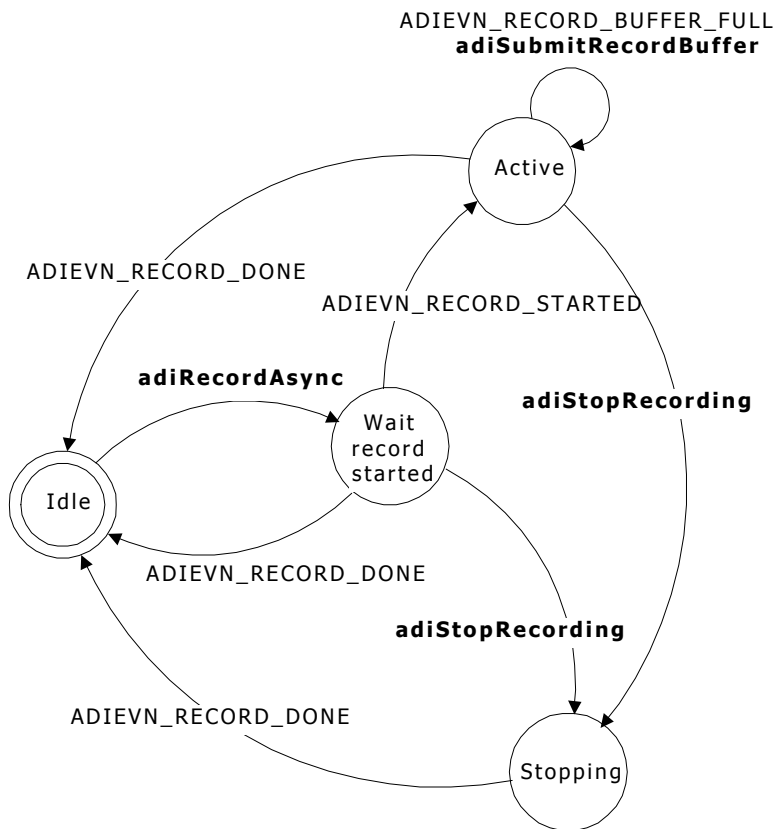
In asynchronous mode, the application transfers voice data from the board to the host by cooperatively exchanging commands and events with the ADI service, as shown in the following illustration:



Transferring voice data during record follows this process:

1. The application initiates recording in asynchronous mode by invoking **adiRecordAsync**.
2. The ADI service generates ADIEVN_RECORD_STARTED to inform the application to submit the second buffer.
3. The application submits the buffer by invoking **adiSubmitRecordBuffer**.
4. The ADI service sends ADIEVN_RECORD_BUFFER_FULL to the application when a record buffer has been filled. The buffer address and size are provided.
5. If the ADI_RECORD_BUFFER_REQ bit is set in the value field in ADIEVN_RECORD_BUFFER_FULL, the ADI service needs another record buffer. In response, the application invokes **adiSubmitRecordBuffer**.
6. Steps 2 - 5 are repeated until recording completes and the ADI service generates ADIEVN_RECORD_DONE.

The following illustration shows the complete life cycle for record using asynchronous data transfer:



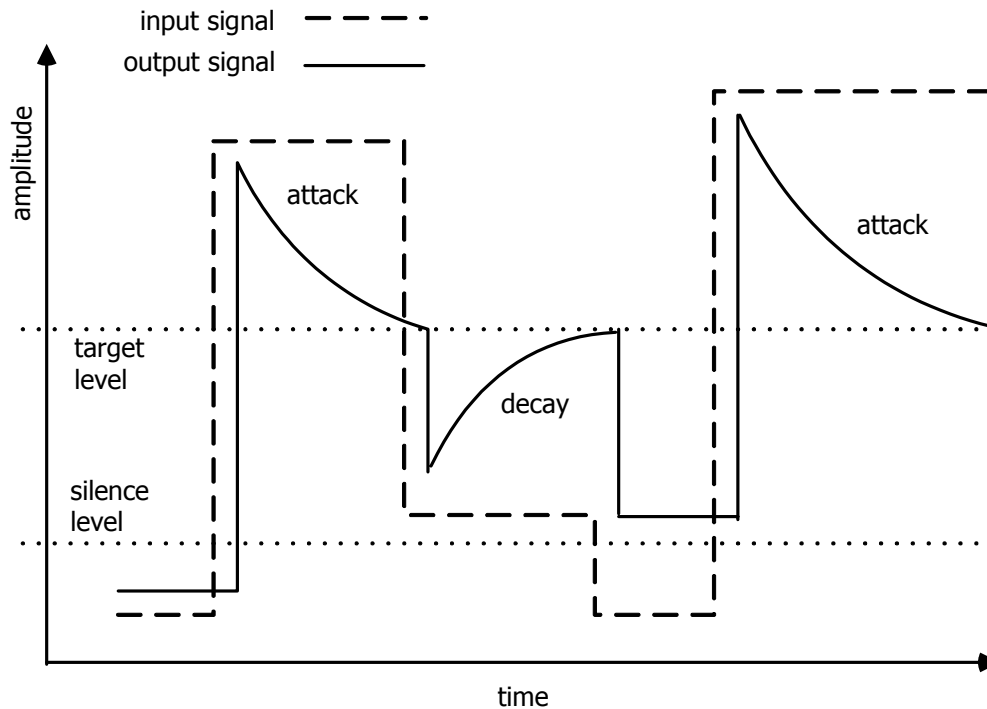
The states for asynchronous record transfer are as follows:

State	Description
Idle	The function is not active.
Wait record started	The record function enters this state when the application invokes adiRecordAsync . The ADI service sends the initial buffer to the board. The board responds with ADIEVN_RECORD_STARTED at which time, the board is actively recording. The application must submit the second required record buffer if the ADI_RECORD_BUFFER_REQ bit is set in the event's value field.
Active	<p>The record function enters the active state after receiving ADIEVN_RECORD_STARTED. The record function remains active until one of the terminating conditions described in <i>Terminating record</i> on page 20 occurs. The ADI service and the application exchange buffer full events and submit buffer commands while in this state as described:</p> <ul style="list-style-type: none"> • The ADI service generates ADIEVN_RECORD_BUFFER_FULL when a record buffer is full. • In response, the application invokes adiSubmitRecordBuffer to continue recording. • A maximum of two user record buffers can be actively submitted at any given time. adiSubmitRecordBuffer returns the error ADIERR_TOO_MANY_BUFFERS if a third buffer is submitted.
Stopped	The application can immediately abort the record function by invoking adiStopRecording . The ADI service does not execute any more record functions from the application while in the stopping state. Any record functions invoked by the application result in the ADI service returning CTAERR_INVALID_SEQUENCE. When ADIEVN_RECORD_DONE is delivered to the application, the record state returns to idle.

Recording with automatic gain control

By default, AGC is disabled and the record gain is determined only by the gain parameter. To enable AGC, set AGCenable in ADI_RECORD_PARMS to 1.

The following illustration shows the automatic gain control (AGC) record parameters:



AGCtargetampl, AGCsilenceampl, AGCattacktime, and AGCdecaytime control the behavior of the AGC. The default values for these parameters are appropriate for most applications. Refer to *ADI_RECORD_PARMS* on page 262 for a description of each of the AGC parameters.

Note: When AGC is enabled, the gain parameter in ADI_RECORD_PARMS determines the gain applied when record begins. AGC must be disabled if you are using voice activity detection.

Playing

Playing follows this process:

1. The application invokes a function to initiate playing.
2. The ADI service prompts the application for data.
3. The application provides data to the ADI service and can instruct the ADI service to automatically stop playing after the buffer plays (by setting the ADI_LASTBUFFER_SUBMITTED flag).
Steps 2 and 3 are typically performed multiple times.
4. The ADI service terminates play upon delivering ADIEVN_PLAY_DONE. Refer to *Terminating play* on page 26 for termination reasons that can be included as part of the event.

The ADI_PLAY_PARMS structure contains the play function parameters.

Initiating play

The ADI service provides three functions to initiate playing speech. The function used depends upon the data transfer method selected:

Use this function...	When the...
adiPlayFromMemory	Application submits a single memory buffer to the ADI service.
adiStartPlaying	ADI service invokes application callback when data is needed. Note: Applications running in client/server mode do not support adiStartPlaying .
adiPlayAsync	ADI service generates a buffer request event when more data is needed. The application asynchronously submits play buffers in response.

The ADI service returns SUCCESS if the start playing command is successfully sent to the board.

Terminating play

The play function terminates when the ADI service delivers ADIEVN_PLAY_DONE, regardless of the transfer method selected. The event value field contains the termination reason, as follows:

If...	Then play ends with...
The application submitted a buffer with the ADI_LASTBUFFER_SUBMITTED flag and the buffer finished playing	CTA_REASON_FINISHED
The call was released by either party	CTA_REASON_RELEASED
A DTMF digit specified in the abort mask was entered by the remote party	CTA_REASON_DIGIT
The application aborted play by calling adiStopPlaying	CTA_REASON_STOPPED
The play was aborted by the speech recognizer	CTA_REASON_RECOGNITION

Playing voice data in callback mode

In callback mode, the ADI service allocates a buffer and invokes an application-specified function to play voice data into it. You specify the callback function when play is initiated with **adiStartPlaying**.

When the ADI service requires data, it invokes the callback function, passing it a buffer to fill and the buffer size. The application's callback routine reads data from a storage medium (for example, a disk) into the buffer. The callback returns the amount of data read and a flag indicating whether to terminate the playing session after the buffer is played.

Playing voice data using callback mode follows this process:

1. The application invokes **adiStartPlaying**.
The ADI service invokes the callback function from within the **adiStartPlaying** context to retrieve the initial buffer (before **adiStartPlaying** returns).
2. The ADI service invokes the application's callback function when a play buffer needs to be filled with voice data.

3. The application's callback function fills the buffer before returning.

At this point, if the application indicates that this is the last buffer (using the `ADI_LASTBUFFER_SUBMITTED` flag) or if a termination condition occurs, the play operation may terminate.

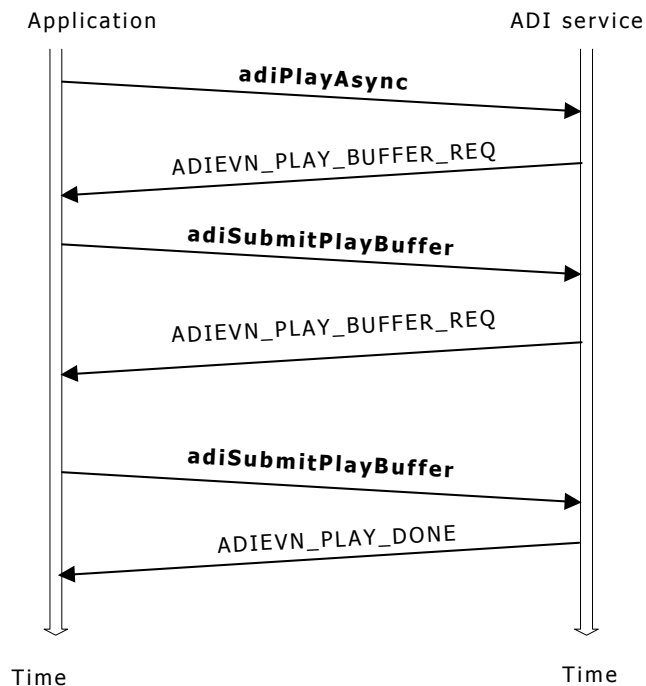
4. Steps 2 and 3 are repeated until the ADI service generates `ADIEVN_PLAY_DONE`.

The application cannot invoke ADI service functions while the callback is executing.

Delaying the callback function could interfere with event processing for any context opened on the same queue.

Playing voice data in asynchronous mode

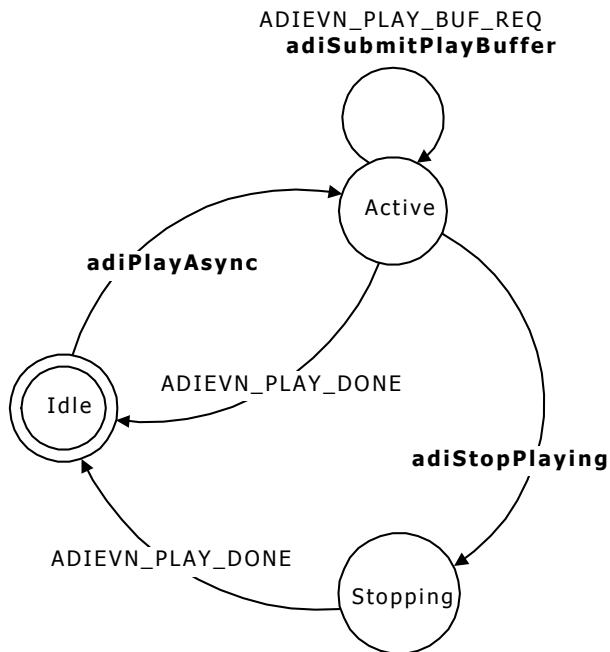
In asynchronous mode, the application transfers voice data from the host to the board by cooperatively exchanging commands and events with the ADI service, as shown:



Transferring voice data asynchronously during play follows this process:

1. The application invokes **`adiPlayAsync`**.
2. The ADI service sends `ADIEVN_PLAY_BUFFER_REQ` whenever the board starts a new buffer.
3. The application invokes **`adiSubmitPlayBuffer`** in response to `ADIEVN_PLAY_BUFFER_REQ`.
4. Steps 2 and 3 are repeated until play completes and the ADI service generates `ADIEVN_PLAY_DONE`.

The following illustration shows the life-cycle for play in asynchronous transfer mode:



The three states for asynchronous play transfer are:

State	Description
Idle	Play is not active.
Active	<p>When the application invokes adiPlayAsync, the ADI service sends the initial buffer to the board and transits to the active state. The play state remains active until one of the terminating conditions described in <i>Terminating play</i> on page 26 occurs.</p> <p>The ADI service sends events and the application submits buffers while in this state as described:</p> <ul style="list-style-type: none"> The ADI service generates ADIEVN_PLAY_BUFFER_REQ whenever the board starts a new buffer (more play data is needed). In response to the ADI service, the application invokes adiSubmitPlayBuffer to continue playing. The application can terminate the play function by setting the ADI_LASTBUFFER_SUBMITTED flag. The ADI service generates ADIEVN_PLAY_DONE when the data already submitted has been played. <p>The application cannot invoke adiSubmitPlayBuffer unless the ADI service has given it ADIEVN_PLAY_BUFFER_REQ. The ADI service returns ADIERR_TOO_MANY_BUFFERS when adiSubmitPlayBuffer is invoked without first receiving a buffer request event.</p>
Stopping	<p>The application can abort play by invoking adiStopPlaying. The ADI service does not accept more play commands from the application while in the stopping state. Any play functions invoked by the application prompt the ADI service to return CTAERR_INVALID_SEQUENCE. When ADIEVN_PLAY_DONE is delivered to the application, the play state returns to idle.</p>

Controlling gain during play

Adjust the play volume at play initiation by changing the default value of the play gain parameter stored in `ADI_PLAY_PARMS` on page 261. You can also modify volume at any time while the play function is active by calling **adiModifyPlayGain**. The default value of the gain is 0 dB (no gain). Gain can be set to any value in the range of -54 dB to +24 dB.

Controlling speed during play

The playing speed can also be adjusted for some encodings. To modify the play speed, call **adiModifyPlaySpeed** during a currently active play. Speed control is available for the following encoding formats:

- `ADI_ENCODE_NMS_16`
- `ADI_ENCODE_NMS_24`
- `ADI_ENCODE_NMS_32`
- `ADI_ENCODE_NMS_64`
- `ADI_ENCODE_OKI_24`
- `ADI_ENCODE_OKI_32`

If you invoke **adiModifyPlaySpeed** for a play operation with data in any other encoding format, the play operation continues at its original speed.

Note: The PacketMedia HMP process and the QX 2000 board do not support **adiModifyPlaySpeed**.

To enable speed control, increase the maxspeed play parameter stored in `ADI_PLAY_PARMS` from its default value of 100.

When play is started with a higher value of maxspeed, the necessary DSP resources are allocated to support increased speed. You can start play with a fast speed (up to maxspeed) by changing the value of the speed parameter in the function call. For the AG boards and the CG boards, slow down up to 50 percent of normal speed is supported.

Note: Starting play with maxspeed greater than 100 requires additional DSP resources beyond that required for playing at normal speed. To determine whether your boards and configuration can support speed up, refer to the *NMS OAM System User's Manual*.

System restrictions

Consider the following system restrictions when using voice record and playback:

- Only one function can drive the output to the telephone line; therefore, the following functions are mutually exclusive:
 - Voice recording with beep enabled (since the record beep prompt drives the output)
 - Voice play
 - Tone generation
 - FSK sending
- If the DTMF detector is disabled, voice functions cannot terminate when digits are entered.
- The following functions are typically configured to share the same group of task processors:
 - Call progress
 - Voice record
 - Voice play
 - Tone generation
 - MF detection

For the typical configuration, DSP capacity is allotted under the assumption that every context is running no more than one of these functions at any given time. There is nothing preventing the application from concurrently executing some combinations of these functions on some contexts. If, however, multiple contexts concurrently execute a combination of these functions, the DSP capacity may be exhausted.

Delays in data processing

NMS Communications boards support DSP functions using a variety of data block sizes. As a consequence, the delays in data processing depend on the data block size of the specific DSP function. In addition, command and event processing to and from the DSPs on these boards occurs at a rate faster than 10 ms.

Using simultaneous play and record

To use simultaneous play and record with an AG board, add the following line to the board section in the board keyword file:

```
Buffers[0].Num=n
```

where *n* = 4 times the number of ports on your board. For example, an AG 2000 board contains 8 ports, so *n* would be 32.

You must disable the beep when recording. If you do not, the record function tries to seize the output and generates CTAERR_OUTPUT_ACTIVE. To disable the beep, set the record parameter beepfreq or beetime to 0.

Performing NMS native play and record

NMS native play and record enables applications to maintain the quality of audio data played and recorded over network interfaces while minimizing the encoding and decoding resources needed to process the audio data.

Applications can perform the following tasks with NMS native play and record:

- Record voice data from RTP data streams transferred through MSPP service endpoints.
- Play and record media streams that contain silence, SID frames, RFC 2833 markers, and lost frame markers.
- Perform silence and DTMF detection while recording RTP streams.
- Play media recorded directly from RTP streams to PSTN (DS0) ports without changing the native format of the data.

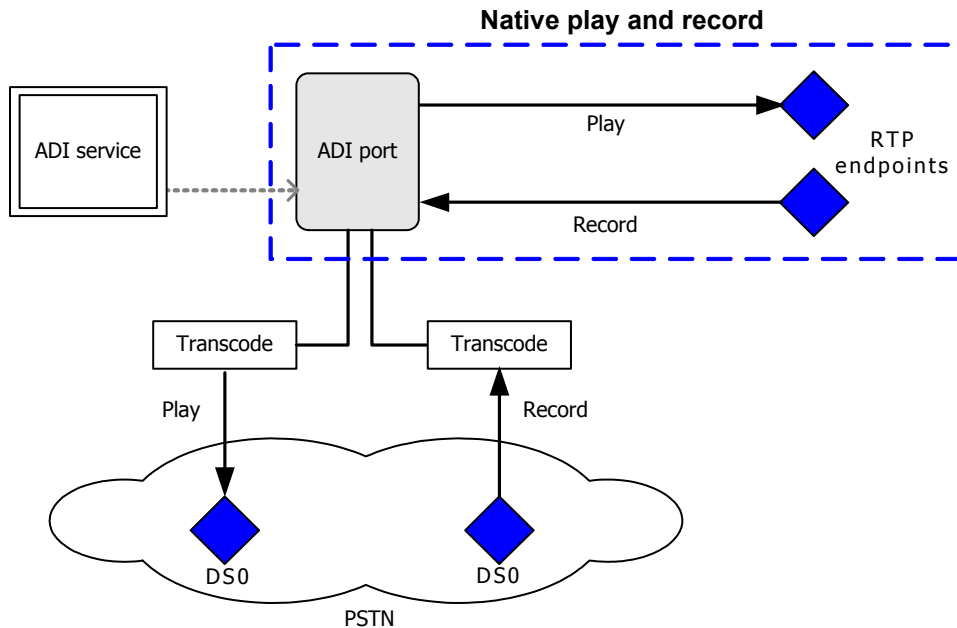
CG boards and PacketMedia HMP processes support NMS native play and record functionality.

NMS native play and record advantages

When an application plays or records audio data over an IP network, typically the application must encode or decode the data. Audio data is often encoded in a compressed format such as G.711 or G.723.1. Encoding or decoding the audio stream can consume system resources and incrementally degrade the quality of the data.

When an application records audio data using native record, the audio is stored in the NMS EDTX (extended discontinuous transmission) format without encoding the data. The application can then either play back the audio data directly to a network interface or transfer the data to the PSTN interface through a decoder.

The following illustration shows the advantages of native play and record:



Implementing NMS native play and record

NMS native play and record uses an NMS proprietary format called EDTX (extended discontinuous transmission) to store and play back codec frames. EDTX formatting incorporates an optional silence compression scheme that uses silence frames in the recorded stream to indicate periods of silence.

Native play and record supports the following encoding types:

- AMR (CG boards only)
- G.711A, G.711U
- G.723.1
- G.726
- G.729A/B

For more information about supported vocoder types, refer to the Fusion vocoder *readme.txt* files.

Applications can implement native play and record in the following ways:

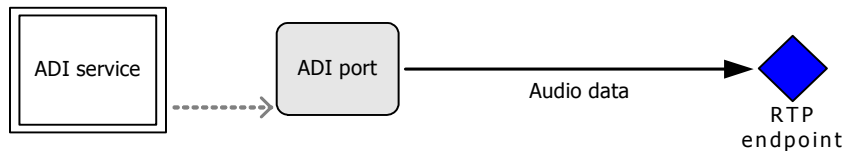
Implementation	Description
Native play	Application plays a stream of audio data from an ADI port to an RTP endpoint.
Native record without inband silence and DTMF detection	Application receives and records a stream of audio data from an RTP endpoint. No data decoding takes place, inband silence detection is not supported, and DTMF detection is supported only through Fusion RFC 2833 support.
Native record with inband silence and DTMF detection	Application receives and records a stream of audio data from an RTP endpoint, and in parallel, decodes the data from its network format (for example, G.711A or G.723.1). The application also performs silence detection, DTMF detection, or both with the data. For a PacketMedia HMP process, refer to the <i>PacketMedia HMP Developer's Manual</i> for implementation procedures.

Native play

To implement native play functionality, the application performs the following tasks:

- Opens the ADI service on the context and starts the nocc protocol.
- Opens the MSPP service on the context and creates an RTP endpoint.
- Retrieves the filter ID of the RTP endpoint.
- Supplies the ADI service with information about the RTP audio streams and specifies the desired behavior for native play operations.
- Starts and stops playing audio data from a native audio stream.

The following illustration shows an overview of the native play mechanism:



Sample procedure

When implementing native play functionality, applications use functions from the following resources:

- Natural Access functions to set up event queues and contexts, and to open services on the contexts.
- ADI service functions to start a protocol, set native play settings, and play out audio data.
- MSPP functions to create an RTP endpoint and retrieve the unique filter ID for the endpoint.

The following procedure shows functions used to implement a typical native play operation:

Step	Action
1	Invoke ctaCreateQueue to create a Natural Access event queue. <code>ctaCreateQueue (&queuehd)</code>
2	Invoke ctaCreateContext to create a Natural Access context for the audio channel. <code>ctaCreateContext (queuehd, &ctahd)</code>
3	Invoke ctaOpenServices to open the ADI and MSPP services on the context. <code>ctaOpenServices (ctahd, svclist, nsvcs)</code>
4	Invoke adiStartProtocol to start the nocc protocol on the ADI port. <code>adiStartProtocol (ctahd, "nocc", NULL, startparms)</code>
5	Invoke mspCreateEndpoint to create an audio MSPP service RTP endpoint. mspCreateEndpoint returns an MSPP service endpoint handle (ephd). <code>mspCreateEndpoint (ctahd, mspaddrstruct, mspparmstruct, &rtpephd)</code>
6	Invoke mspGetFilterHandle to retrieve the runtime filter ID (fltID) associated with the RTP endpoint handle (ephd). The application uses the returned fltID as the destination for the audio stream played out from the ADI port. <code>mspGetFilterHandle (rtpephd, MSP_ENDPOINT RTPFDX, &fltID)</code>
7	Invoke adiSetNativeInfo to set NMS native play parameters, specifying both the context handle of the ADI port and the RTP endpoint fltID returned by mspGetFilterHandle . <code>adiSetNativeInfo (ctahd, NULL, fltID, fltID_parms)</code>
8	Invoke adiPlayFromMemory to begin playing a message. <code>adiPlayFromMemory (ctahd, encoding, buffer, bufsize, parms)</code>
9	Invoke adiStopPlaying to stop playing the message. <code>adiStopPlaying (ctahd)</code>

Example

The following example shows how to perform a native play operation:

```
ret = ctaCreateQueue( NULL, 0, &hCtaQueueHd );

ret = ctaCreateContext( hCtaQueueHd, 0, "Play", &ctahd );

ServiceCount = 2;
ServDesc[0].name.svcname      = "ADI";
ServDesc[0].name.svcmgrname   = "ADIMGR";
ServDesc[0].mvipaddr.board    = board;
ServDesc[0].mvipaddr.mode     = 0;
ServDesc[1].name.svcname      = "MSP";
ServDesc[1].name.svcmgrname   = "MSPMGR";
ret = ctaOpenServices( ctahd, ServDesc, ServiceCount );
ret = WaitForSpecificEvent( CTAEVN_OPEN_SERVICES_DONE, &event );

ret = adiStartProtocol( ctahd, "nocc", NULL, NULL );
ret = WaitForSpecificEvent( ADIEVN_STARTPROTOCOL_DONE, &event );

// create mspp RTP endpoint
ret = mspCreateEndpoint( ctahd, &mspAddr, &mspParm, &ephid );
ret = WaitForSpecificEvent( MSPEVN_CREATE_ENDPOINT_DONE, &event );

// get cg6xxx board handle
ret = mspGetFilterHandle( mspHd, MSP_FILTER RTPFDX_EPH, rtp_play_filter_handle );
ret = adiSetNativeInfo( ctahd, NULL, /* no ingress handle, as this is a play only */
rtp_play_filter_handle, &natpr_ctl ); /* RTP endpoint filter ID
specified as a destination for audio */
ret = adiPlayFromMemory( ctahd, ADI_ENCODE_EDTX_AMRNB, /* audio play */
MemoryBuffer, RecordedBytes, NULL );
.
.
.
ret = adiStopPlaying ( ctahd );
```

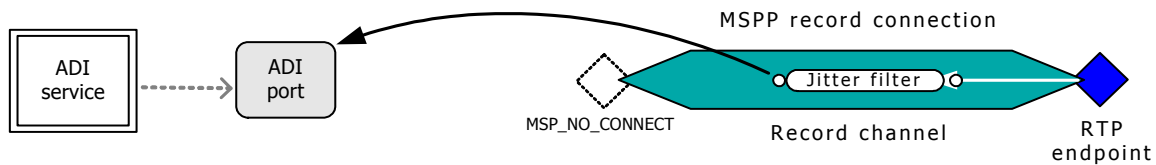
Native record without inband silence and DTMF detection

To implement native record functionality without inband silence detection or DTMF detection, the application performs the following tasks:

- Creates a Natural Access event queue and context.
- Opens the ADI and MSPP services on the context and starts the nocc protocol on the context.
- Creates an MSPP RTP endpoint and an MSPP record channel on the context.
- Connects the RTP endpoint with the record channel to create a record connection.
- Retrieves the filter ID of the jitter filter within the record channel.
- Supplies the ADI service with information about the RTP audio streams and specifies the desired behavior for native record operations.
- Starts and stops recording audio data from a network audio stream.

Note: Applications can perform DTMF detection using Fusion RFC 2833 support, but silence detection is not supported. Refer to the *Fusion Developer's Manual* for more information.

The following illustration shows an overview of the native record mechanism without voice decoding:



Sample procedure

Applications use functions from the following Natural Access resources to implement native record functionality without inband silence detection or DTMF detection:

- Natural Access functions to set up event queues and contexts, and to open services on the contexts.
- ADI service functions to start a protocol, set native record settings, and record incoming audio data.
- MSPP functions to create a voice connection consisting of a record channel and an RTP endpoint, and to retrieve the unique filter ID associated with the record channel.

The following procedure shows functions used to implement a typical native record operation without decoding:

Step	Action
1	Invoke ctaCreateQueue to create a Natural Access event queue. <code>ctaCreateQueue (&queuehd)</code>
2	Invoke ctaCreateContext to create a Natural Access context for the audio channel. <code>ctaCreateContext (queuehd, &ctahd)</code>
3	Invoke ctaOpenServices to open the ADI and MSPP services on the context. <code>ctaOpenServices (ctahd, svclist, nsvcs)</code>
4	Invoke mspCreateEndpoint to create an audio RTP endpoint. mspCreateEndpoint returns an endpoint handle (<i>eph</i>). <code>mspCreateEndpoint (ctahd, mspaddrstruct, msparmstruct, &rtpephd)</code>
5	Invoke mspCreateChannel to create a record channel. <code>mspCreateChannel (ctahd, chnladdr, chnlparms, &chanhd)</code>
6	Invoke mspConnect to connect the record channel with the RTP endpoint. Specify MSP_NO_CONNECT instead of a DS0 endpoint handle. <code>mspConnect (MSP_NO_CONNECT, chanhd, rtpephd)</code>
7	Invoke mspEnableChannel to enable the record channel to process data. <code>mspEnableChannel (msphd)</code>
8	Invoke adiStartProtocol to start the nocc protocol on the audio channel. <code>adiStartProtocol (ctahd, "nocc", NULL, startparms)</code>
9	Invoke mspGetFilterHandle to retrieve the filter identifier (<i>fltID</i>) associated with the MSPP record channel. <code>mspGetFilterHandle (chanhd, MSP_FILTER_JITTER, &fltID)</code>

Step	Action
10	Invoke adiSetNativeInfo to set NMS native record parameters. Specify both the context handle of the ADI port and the <i>fltID</i> returned by mSPGetFilterHandle . <code>adiSetNativeInfo (ctahd, fltID, NULL, natpr_parms)</code>
11	Invoke ctaGetParms to return parameter values for the ADI_RECORD_PARMS structure. <code>ctaGetParms(ctahd, parmId, buffer, size)</code>
12	Invoke adiRecordToMemory to begin recording audio data. <code>adiRecordToMemory (ctahd, buf, bufsize, rec_param)</code>
13	Invoke adiStopRecording stop recording audio data. <code>adiStopRecording (ctahd)</code>

Example

The following example shows how to perform a native record operation without decoding:

```
ret = ctaCreateQueue( NULL, 0, &hCtaQueueHd );
ret = ctaCreateContext( hCtaQueueHd, 0, "Record", &ctahd );

ServiceCount = 2;
ServDesc[0].name.svcname      = "ADI";
ServDesc[0].name.svcmgrname   = "ADIMGR";
ServDesc[0].mvipaddr.mode     = ADI_VOICE_DUPLEX;
ServDesc[0].mvipaddr.stream   = 0;
ServDesc[0].mvipaddr.timeslot = record_timeslot;
ServDesc[1].name.svcname      = "MSP";
ServDesc[1].name.svcmgrname   = "MSPMGR";

ret = ctaOpenServices( ctahd, ServDesc, ServiceCount );
ret = WaitForSpecificEvent( CTAEVN_OPEN_SERVICES_DONE, &Event );

// IP Channel Initialization
MSPHD ds0_ephd = MSP_NO_CONNECT;
MSPHD rtp_ephd;

// Create and init RTP endpoint
&
mspCreateEndpoint( ctaHd, &rtpaddr, &rtp_params, rtp_ephd );
if (! WaitForSpecificEvent( MSPEVN_CREATE_ENDPOINT_DONE, &Event, 5000 ))
{
    printf("Failed waiting for MSPEVN_CREATE_ENDPOINT_DONE (RTP)");
    return FAILURE;
}

chanaddr.nBoard      = Board;
chanaddr.channelType = G711RecordChannel;
chanaddr.FilterAttribs = MSP_FCN_ATTRIB_RFC2833;
chan_params.size     = sizeof( MSP_CHANNEL_PARAMETER );
chan_params.channelType = G711RecordChannel;
chan_params.ChannelParms.VoiceParms.size = sizeof( MSP_VOICE_CHANNEL_PARMS );

// Create channel
mspCreateChannel( ctaHd, &chanaddr, &chan_params, &mSPHD );
CTA_EVENT CtaEvent;
if (! WaitForSpecificEvent( MSPEVN_CREATE_CHANNEL_DONE, &Event, 5000 ))
{
    printf("Failed waiting for MSPEVN_CREATE_CHANNEL_DONE");
    return FAILURE;
}
```

```

// connect mspp endpoints
ret = mspConnect( ds0_ephd, msphd, rtp_ephd );
if (! WaitForSpecificEvent(MSPEVN_CONNECT_DONE, &Event, 5000))
{
    printf("Failed waiting for MSPEVN_CONNECT_DONE");
    return FAILURE;
}

// connect mspp enable channel
mspEnableChannel( msphd );
if (! WaitForSpecificEvent(MSPEVN_ENABLE_CHANNEL_DONE, &Event, 5000))
{
    printf("Failed waiting for MSPEVN_ENABLE_CHANNEL_DONE");
    return FAILURE;
}

//adiStartProtocol
adiStartProtocol( ctahd, "nocc", NULL, NULL );
if (! WaitForSpecificEvent( ADIEVN_STARTPROTOCOL_DONE, &Event, 5000))
{
    printf("Failed to receive ADIEVN_STARTPROTOCOL_DONE event");
    return FAILURE;
}

// get cg6xxx board handle
ret = mspGetFilterHandle( msphd, MSP_FILTER_JITTER, &cg6xxx_board_filter_handle );

ADI_NATIVE_CONTROL parms = {0};          /* Native parameters */
parms.frameFormat          = 0;
parms.include2833          = 0;
parms.vadFlag              = 0;
parms.nsPayload             = 0;
parms.mode                  = ADI_NATIVE;
parms.rec_encoding          = ADI_ENCODE_EDTX_MU_LAW;
parms.payloadID             = 0;
ret = adiSetNativeInfo( ctahd, cg6xxx_board_filter_handle,
NULL, /* this is record only so no egress handle */
&parms);

// get default adi record parms
ret = ctaGetParms( ctahd, ADI_RECORD_PARMID, &recparms, sizeof(ADI_RECORD_PARMS) );
ret = adiRecordToMemory( ctahd, ADI_ENCODE_EDTX_MU_LAW, /* audio rec */
MemoryBuffer, RecordedBytes, &recparms );
.
.
.
adiStopRecording ( ctahd );

```

Native record with inband silence and DTMF detection

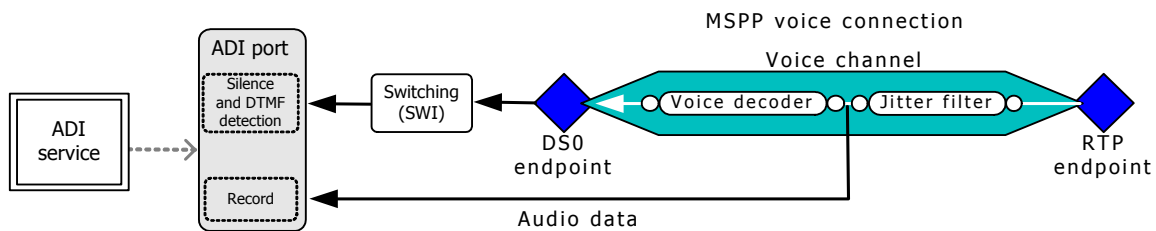
On CG boards, use the following procedure to implement native record with inband silence detection or DTMF detection. For PacketMedia HMP processes, refer to the implementation procedure in the *PacketMedia HMP Developer's Manual*.

To implement native record functionality with inband silence detection or DTMF detection, the application performs the following tasks:

- Opens the ADI service on a Natural Access context and starts the nocc protocol on the context.
- Opens the MSPP service and the ADI service on a second context, and creates an RTP endpoint, a DS0 endpoint, and a voice channel on the context.
- Connects the RTP endpoint, DS0 endpoint, and voice channel to create a voice connection.
- Creates a switch connection between the ADI port and the DS0 endpoint.
- Retrieves the filter ID of the jitter filter associated with the voice channel.

- Supplies the ADI service with information about the RTP audio streams and specifies the desired behavior for native record operations.
- Starts and stops recording audio data from a network audio stream.

The following illustration shows an overview of the native record mechanism with voice decoding enabled:



Sample procedure

Applications use functions from the following Natural Access resources to implement native record functionality with inband silence detection or DTMF detection:

- Natural Access functions to set up event queues and contexts and to open services on the contexts.
- ADI service functions to start a protocol, set native record settings, and record incoming audio data.
- MSPP functions to create a voice connection consisting of a voice decoding channel, an RTP endpoint, and a DS0 endpoint, and to retrieve the unique filter ID of the RTP endpoint's jitter filter.
- SWI functions to switch together the ADI service port and the MSPP service connection (through the DS0 endpoint).

The following procedure shows functions used to implement a typical native record operation with decoding on CG boards:

Step	Action
1	Invoke ctaCreateQueue to create a Natural Access event queue. <code>ctaCreateQueue (&queuehd)</code>
2	Invoke ctaCreateContext to create a Natural Access context for the audio channel. <code>ctaCreateContext (queuehd, &ctahd)</code>
3	Invoke ctaOpenServices to open the ADI service on the context. When using ctaOpenServices , the application must specify the following: <ul style="list-style-type: none"> • Set the <code>svclist.mvipaddr.mode</code> parameter to <code>ADI_VOICE_DUPLEX</code> to allocate DSP resources for the channel on the CG board. • Set the <code>svclist.mvipaddr.stream</code> parameter to 0 and the <code>svclist.mvipaddr.timeslot</code> parameter to a unique and valid entry. For more information, refer to the <i>Natural Access Developer's Reference Manual</i>. <code>ctaOpenServices (ctahd, svclist, nsvcs)</code>
4	Invoke adiStartProtocol to start the nocc protocol on the audio channel and enable silence detection on the audio channel. <code>adiStartProtocol (ctahd, "nocc")</code>

Step	Action
5	<p>Invoke swiOpenSwitch to open a switching device for the context. swiOpenSwitch returns a switch handle (<i>swihd</i>).</p> <pre>swiOpenSwitch (ctahd, "cg6ksw", board, 0x0, &swihd)</pre>
6	<p>Invoke ctaCreateContext to create a Natural Access context for the MSPP channel.</p> <pre>ctaCreateContext (queuehd, &msphd)</pre>
7	<p>Invoke ctaOpenServices to open the MSPP service on the context.</p> <pre>ctaOpenServices (msphd, svclist, nsvcs)</pre>
8	<p>Invoke mSPCreateEndpoint to create an audio RTP endpoint. mSPCreateEndpoint returns an endpoint handle (<i>&rtpephd</i>).</p> <pre>mSPCreateEndpoint (msphd, mspaddrstruct, mspparmstruct, &rtpephd)</pre>
9	<p>Invoke mSPCreateEndpoint to create an audio DS0 endpoint. mSPCreateEndpoint returns an endpoint handle (<i>&ds0ephd</i>).</p> <pre>mSPCreateEndpoint (msphd, mspaddrstruct, mspparmstruct, &ds0ephd)</pre>
10	<p>Invoke mSPCreateChannel to create a full duplex or voice decoding channel.</p> <pre>mSPCreateChannel (msphd, chnladdr, chnlparms, &chanhd)</pre>
11	<p>Invoke mSPConnect to connect the DS0 and RTP endpoints with the voice channel.</p> <pre>mSPConnect (ds0ephd, chanhd, rtpephd)</pre>
12	<p>Invoke mSPEnableChannel to enable the record channel to process data.</p> <pre>mSPEnableChannel (msphd)</pre>
13	<p>Invoke swiMakeConnection with the <i>swihd</i> returned by swiOpenSwitch to connect the MSPP DS0 output to the ADI audio channel input and vice versa. When using swiMakeConnection, the application specifies the stream and timeslot used to create the ADI port and the stream and timeslot used to create the DS0 endpoint.</p> <pre>swiMakeConnection (swihd, adi_ds0, ds0ephd, 2)</pre>
14	<p>Invoke mSPGetFilterHandle to retrieve the filter identifier (<i>fltID</i>) associated with the MSPP record channel.</p> <pre>mSPGetFilterHandle (chanhd, MSP_FILTER_JITTER, &fltID)</pre>
15	<p>Invoke adiSetNativeInfo to set NMS native record parameters. Specify both the context handle of the ADI port and the <i>fltID</i> returned by mSPGetFilterHandle.</p> <pre>adiSetNativeInfo (ctahd, fltID, NULL, natpr_parms)</pre>
16	<p>Invoke ctaGetParms to return parameter values for the ADI_RECORD_PARMS structure.</p> <pre>ctaGetParms (ctahd, parmId, buffer, size)</pre>
17	<p>Invoke adiRecordToMemory to begin recording a message.</p> <pre>adiRecordToMemory (ctahd, buf, bufsize, rec_param)</pre>
18	<p>Invoke adiStopRecording stop recording the audio portion of the message.</p> <pre>adiStopRecording (ctahd)</pre>

Example

The following example shows how to perform a native record operation that supports ADI silence and DTMF detection on CG boards:

```
ret = ctaCreateQueue( NULL, 0, &hCtaQueueHd );
// create context for ADI port
ret = ctaCreateContext( hCtaQueueHd, 0, "Record", &ctahd );

ServiceCount = 1;
ServDesc[0].name.svcname      = "ADI";
ServDesc[0].name.svcmgrname   = "ADIMGR";
ServDesc[0].mvipaddr.mode     = ADI_VOICE_DUPLEX;
ServDesc[0].mvipaddr.stream   = 0;
ServDesc[0].mvipaddr.timeslot = record_timeslot;

ret = ctaOpenServices( ctahd, ServDesc, ServiceCount );
ret = WaitForSpecificEvent( CTAEVN_OPEN_SERVICES_DONE, &Event );
{
    printf("Failed to receive CTAEVN_OPEN_SERVICES_DONE event");
    return FAILURE;
}

//adiStartProtocol
adiStartProtocol( ctahd, "nocc", NULL, NULL );
if ( ! WaitForSpecificEvent( ADIEVN_STARTPROTOCOL_DONE, &Event, 5000 ) )
{
    printf("Failed to receive ADIEVN_STARTPROTOCOL_DONE event");
    return FAILURE;
}

ret = swiOpenSwitch(ctahd, "agsw", Board, 0, &swihd);
if (ret != SUCCESS)
{
    printf("Makeconnections: Failed to open board %d\n", Board);
    return FAILURE;
}

// create context for MSPP channel
ret = ctaCreateContext( hCtaQueueHd, 0, "MSPP", &ipHd );

ServiceCount = 2;
ServDesc[0].name.svcname      = "ADI";
ServDesc[0].name.svcmgrname   = "ADIMGR";
ServDesc[0].mvipaddr.mode     = ADI_VOICE_DUPLEX;
ServDesc[0].mvipaddr.stream   = 0;
ServDesc[0].mvipaddr.timeslot = fusion_timeslot;
ServDesc[1].name.svcname      = "MSP";
ServDesc[1].name.svcmgrname   = "MSPMGR";

ret = ctaOpenServices( iphd, ServDesc, ServiceCount );
ret = WaitForSpecificEvent( CTAEVN_OPEN_SERVICES_DONE, &Event );
{
    printf("Failed to receive CTAEVN_OPEN_SERVICES_DONE event");
    return FAILURE;
}

// IP Channel Initialization
MSPHD      ds0_ephd;
MSPHD      rtp_ephd;

// Create and init RTP endpoint
MSP_ENDPOINT_ADDR      rtpaddr      = {0};
MSP_ENDPOINT_PARAMETER rtp_params = {0};

rtpaddr.size      = sizeof(MSP_ENDPOINT_ADDR);
rtpaddr.eEpType   = MSP_ENDPOINT_RTFFDX;
rtpaddr.nBoard    = Board;
...
```

```

mspCreateEndpoint( ipHd, &rtpaddr, &rtp_params, &rtp_ephd );
if (! WaitForSpecificEvent(MSPEVN_CREATE_ENDPOINT_DONE, &Event, 5000))
{
    printf("Failed waiting for MSPEVN_CREATE_ENDPOINT_DONE (RTP)");
    return FAILURE;
}

// create mspp DS0 endpoint
MSP_ENDPOINT_ADDR    ds0addr    = {0};
ds0addr.eEpType      = MSP_ENDPOINT_DS0;
ds0addr.nBoard       = Board;
ds0addr.size         = sizeof(MSP_ENDPOINT_DS0);
ds0addr.EP.DS0.nTimeslot = fusion_timeslot;
MSP_ENDPOINT_PARAMETER ds0parms  = {0};
ds0parms.size        = sizeof(DS0_ENDPOINT_PARMS);
ds0parms.eParmType   = MSP_ENDPOINT_DS0;
ds0parms.EP.DS0.media = MSP_VOICE;
mspCreateEndpoint( ipHd, &ds0addr, &ds0parms, &ds0_ephd );
if (! WaitForSpecificEvent(MSPEVN_CREATE_ENDPOINT_DONE, &Event, 5000))
{
    printf("Failed waiting for MSPEVN_CREATE_ENDPOINT_DONE (DS0)");
    return FAILURE;
}

// create mspp Channel
MSP_CHANNEL_ADDR    chanaddr    = {0};
MSP_CHANNEL_PARAMETER chan_params = {0};

chanaddr.nBoard      = Board;
chanaddr.channelType  = G711FullDuplex;
chanaddr.FilterAttribs = MSP_FCN_ATTRIB_RFC2833;
chan_params.size      = sizeof( MSP_CHANNEL_PARAMETER );
chan_params.channelType = G711FullDuplex;
chan_params.ChannelParms.VoiceParms.size = sizeof( MSP_VOICE_CHANNEL_PARMS );
...
// Create channel
mspCreateChannel( ipHd, &chanaddr, &chan_params, &msphd );
CTA_EVENT CtaEvent;
if (! WaitForSpecificEvent(MSPEVN_CREATE_CHANNEL_DONE, &Event, 5000))
{
    printf("Failed waiting for MSPEVN_CREATE_CHANNEL_DONE");
    return FAILURE;
}

// connect mspp endpoints
ret = mspConnect(ds0_ephd, msphd, rtp_ephd);
if (! WaitForSpecificEvent(MSPEVN_CONNECT_DONE, &Event, 5000))
{
    printf("Failed waiting for MSPEVN_CONNECT_DONE");
    return FAILURE;
}

// connect mspp enable channel
mspEnableChannel(msphd);
if (! WaitForSpecificEvent(MSPEVN_ENABLE_CHANNEL_DONE, &Event, 5000))
{
    printf("Failed waiting for MSPEVN_ENABLE_CHANNEL_DONE");
    return FAILURE;
}

// connect Fusion and ADI timeslots to allow Silence and DTMF detection
SWI_TERMINUS input[2];
SWI_TERMINUS output[2];

output[0].bus = MVIP95_LOCAL_BUS;
output[0].stream = BoardStream;
output[0].timeslot = record_timeslot;
output[1].bus = MVIP95_LOCAL_BUS;
output[1].stream = BoardStream;
output[1].timeslot = fusion_timeslot;
input[0].bus = MVIP95_MVIP_BUS;

```

```

input[0].stream = BoardStream+1;
input[0].timeslot = fusion_timeslot;
input[1].bus = MVP95_MVIP_BUS;
input[1].stream = BoardStream+1;
input[1].timeslot = record_timeslot;
swiMakeConnection (swihd, input, output, 2)

// get cg6xxx board handle
ret = mspGetFilterHandle( msphd, MSP_FILTER_JITTER, &cg6xxx_board_filter_handle );

ADI_NATIVE_CONTROL parms = {0};      /* Native parameters */
parms.frameFormat = 0;
parms.include2833 = 0;
parms.vadFlag = 0;
parms.nsPayload = 0;
parms.mode = ADI_NATIVE;
parms.rec_encoding = ADI_ENCODE_EDTX_MU_LAW;
parms.payloadID = 0;
ret = adiSetNativeInfo( ctahd, cg6xxx_board_filter_handle,
NULL, /* this is record only so no egress handle */
&parms );

// get default adi record parms
ret = ctaGetParms( ctahd, ADI_RECORD_PARMID, &recparms, sizeof(ADI_RECORD_PARMS) );
ret = adiRecordToMemory( ctahd, ADI_ENCODE_EDTX_MU_LAW, /* audio rec */
MemoryBuffer, RecordedBytes, &recparms );
.
.
.
adiStopRecording (ctahd);

```

Managing call progress

Call progress functions monitor in-band energy to detect network tones, voice, and modem or fax terminal tones. Call progress functions enable you to manage low-level call control directly. Call progress is affected by the parameters stored in the **ADI_CALLPROG_PARMS** structure.

This topic presents:

- Tone detection
- Call progress tone events
- Call progress voice events
- Call progress termination events
- System restrictions

Tone detection

Call progress functions are automatically invoked when **nccPlaceCall** is specified and turned off when the call reaches a connected state. Once the call is in a connected state, an application can invoke call progress functions and analyze in-band energy as described in the following topics.

Telephone network tone detection

The call progress functions analyze in-band audio to detect the following telephone network signals:

- SIT (special information tone)
- Reorder (fast busy)
- Busy

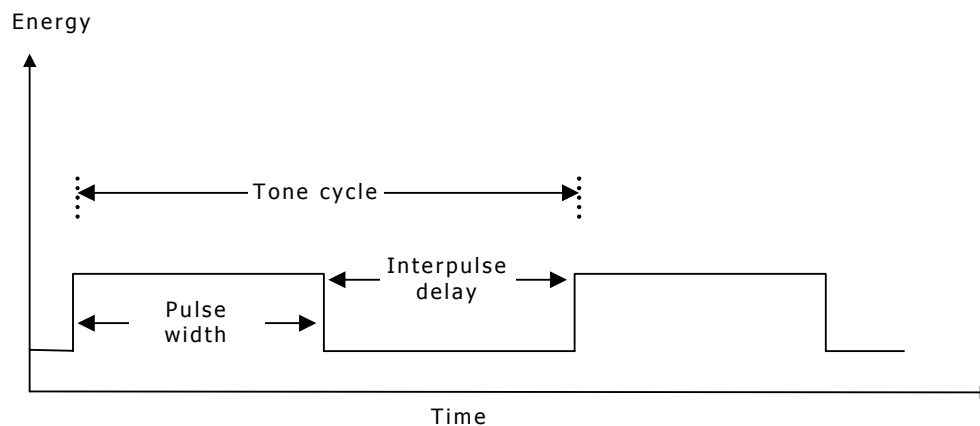
- Ringing (referred to as ring tone)
- Number unassigned tone

The ADI_CALLPROG_PARMS parameters set the criteria to determine if the energy received is a telephone network tone or voice. These parameters are sent to the board by **adiStartCallProgress**.

The following terms are used to characterize telephone network tones:

Term	Description
Pulse width	Time during which a tone is active.
Inter-pulse delay	Time between two active tone pulses.
Tone cycle	Time during which a tone is active and then absent.

The following illustration shows generic tone characteristics:



The ADI service uses a precise tone detector and a broadband tone detector to distinguish tones from voice data.

Precise tone detection

The precise tone detector analyzes in-band audio at specific frequencies to detect the following types of tones:

- Busy
- Reorder
- SIT (special information tone)
- CED (generated by fax terminals or modems)
- TDD/TTY (generated by devices for the hearing impaired)
- Number unassigned

The application specifies which of these tones to detect by configuring the precmask in ADI_CALLPROG_PARMS. If the busy tone detection is not enabled, the ADI service takes more time to discern the busy tone using broadband tone detection. If SIT detection and CED detection are not enabled, these tones cannot be detected.

The precqualtime parameter determines the duration in which the tones are qualified. This parameter applies to all three tones defined in this topic. Set it to the time required to detect the tone of shortest duration.

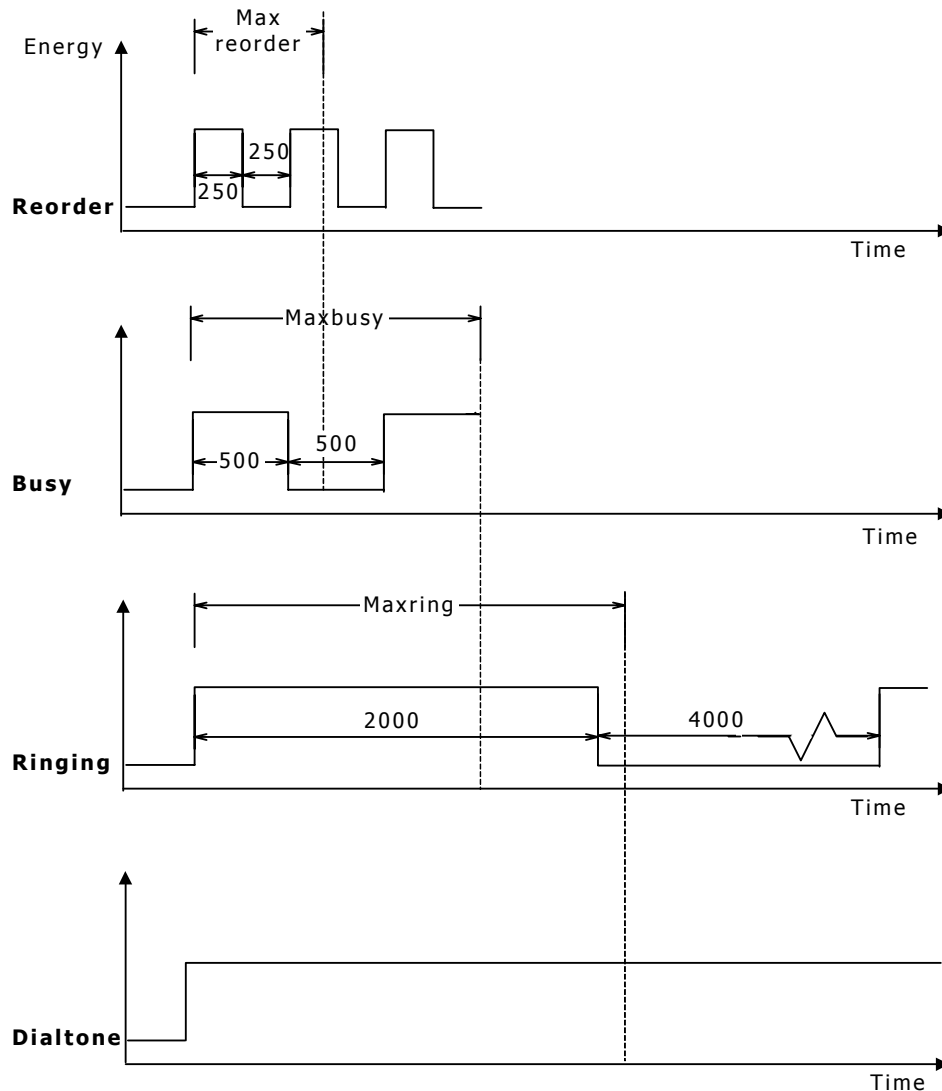
Broadband tone detection

The broadband detection method uses simple high and low pass frequency discrimination together with broadband energy measurements over time to qualify network tones. For example, the DSP determines that a ring tone is present if all of the following conditions are met:

- There is energy below the telephone network tone frequency threshold (1 kHz).
- There is little or no energy above the network tone frequency threshold.
- The amplitude and frequency are reasonably steady over the period of time defining the ring tone.

Two signal characteristics are used for broadband tone detection: time period and cadence.

When defining time period, the application specifies time limits for excluding each telephone network tone. The following illustration shows the effect these limits have on the tones. The waveforms depicted are in milliseconds and are representative of tones in the USA.



The time parameters shown in the previous illustration are stored in `ADI_CALLPROG_PARMs` on page 254.

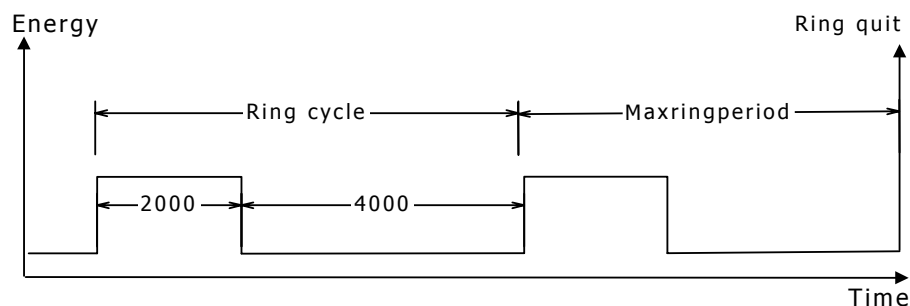
Note: The ADI service presumes that `maxreorder` is less than `maxbusy`, and `maxbusy` is less than `maxring`. This relationship defines a time tolerance (minimum and maximum) for each of the three tones detected.

The second characteristic used for broadband tone detection is the signal's cadence. The application specifies tone counts in `ADI_CALLPROG_PARMs`. The signal must satisfy the single tone criteria described in the following illustration for the respective number of cycles before the ADI service concludes the signal is present. The following table defines the cadence for each signal:

Tone	Parameter	Description
Busy	busycount	Busy signal received.
Reorder	busycount	Reorder received.
Ring	ringcount	Call not answered.

For example, when `busycount` reorder tones are counted, the ADI service concludes it is receiving a reorder (fast busy) signal.

The following illustration depicts ring tone termination. After having established that the line is receiving a ring tone, the ADI service concludes that the remote trunk has quit ringing if a ring tone is not received in the `maxringperiod`. This parameter controls the ring quit event.



Voice detection

If the ADI service does not detect a network tone, call analysis advances into the final stage of voice detection.

The ADI service detects when the remote party starts and stops speaking. These are the voice begin and voice end conditions, respectively. The voice begin condition indicates that the call is being answered by the remote party.

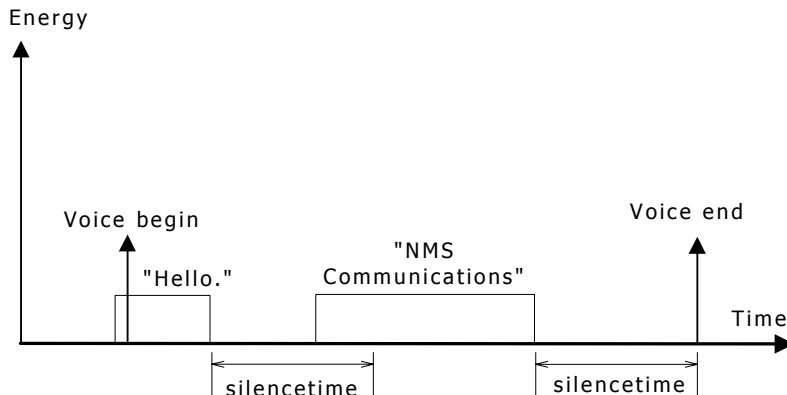
The ADI service supports three voice duration time thresholds: medium, long, and extended. The duration for these three thresholds is specified in the `ADI_CALLPROG_PARMs` structure. The application can set the `connectmask` and `disconnectmask` to force call resolution on any of these voice thresholds, as well as on the voice end condition.

For example, the application expects a voice to begin speaking with a short salutation (for example, Hello). The voice medium time threshold should be set accordingly.

The precise definition of each of these conditions is controlled by parameters in the callprog structure in ADI_CALLPROG_PARMS:

Voice condition	Description	Controlling parameter in callprog in ADI_CALLPROG_PARMS
Voice begin	Remote party begins speaking.	None.
Voice medium	Remote party has spoken for a period longer than the first time threshold.	voicemedium - (ms) first time threshold.
Voice long	Remote party has spoken for a period longer than the second time threshold.	voicelong - (ms) second time threshold.
Voice extended	Remote party has spoken for a period longer than the third time threshold.	voicextended - (ms) third and final time threshold.
Voice end	Remote party stopped speaking.	silencetime - (ms) qualification time before concluding voice end.

Voice begin can be triggered when the remote party begins speaking. Voice end occurs after an absence of voice for silencetime milliseconds. The following illustration shows call progress analysis voice detection:



Using call placement timeout

To ensure that call placement is resolved within a certain time period, the ADI service provides a timeout parameter. The timeout parameter in ADI_CALLPROG_PARMS specifies the maximum time after the last detected event before the ADI service generates ADIEVN_CP_DONE with a value of CTA_REASON_TIMEOUT. Setting the timeout parameter in ADI_CALLPROG_PARMS to zero overrides the timeout feature.

Call progress tone events

The call progress tone events are mapped from tone events described in:

- Telephone network tone detection
- Precise tone detection
- Broadband tone detection

Call progress tone events are controlled by the ADI_CALLPROG_PARMS structure. The following tones are detected by call progress:

If the detected tone is...	The ADI event is...
Dial tone	ADIEVN_CP_DIALTONE
Reorder tone	ADIEVN_CP_REORDERTONE
Ring tone	ADIEVN_CP_RINGTONE
Ring quit	ADIEVN_CP_RINGQUIT
SIT	ADIEVN_CP_SIT
Fax/modem answer tone	ADIEVN_CP_CED
TDD/TTY tone	ADIEVN_CP_TDD

Call progress voice events

The call progress voice events are mapped from the voice events described in *Voice detection* on page 46 and are controlled by the ADI_CALLPROG_PARMS structure.

Whenever a voice event occurs during call progress, ADIEVN_CP_VOICE is generated. The event value field contains the voice event:

If the remote party...	The ADI event reason is...
Begins speaking (voice begin)	ADI_CP_VOICE_BEGIN
Has spoken for a period longer than the first time threshold (voice medium)	ADI_CP_VOICE_MEDIUM
Has spoken for a period longer than the second time threshold (voice long)	ADI_CP_VOICE_LONG
Has spoken for a period longer than the third time threshold (voice extended)	ADI_CP_VOICE_EXTENDED
Stopped speaking (voice end)	ADI_CP_VOICE_END

Call progress termination events

Call progress terminates when any of the following events occur:

If...	The ADI event is...
A dial tone is detected	ADIEVN_CP_DIALTONE
A busy tone is detected	ADIEVN_CP_BUSYTONE
A reorder tone is detected	ADIEVN_CP_REORDERTONE
A SIT tone is detected	ADIEVN_CP_SIT
There is no answer	ADIEVN_CP_NOANSWER
A fax or modem answer tone is detected	ADIEVN_CP_CED
A TDD/TTY device tone is detected	ADIEVN_CP_TDD

Additionally, you can configure the stopmask parameter in the ADI_CALLPROG_PARMS structure to selectively terminate on the occurrence of any of the following telephone network events:

Telephone network event	ADI event
A ring tone is detected.	ADIEVN_CP_RINGTONE
There is a loss of ring tone with no subsequent events.	ADIEVN_CP_RINGQUIT
Remote party begins speaking (voice begin). Check the value field of the event for the voice event.	ADIEVN_CP_VOICE, with ADIEVN_CP_VOICE_BEGIN in the value field
Remote party has spoken for a period longer than the first time threshold (voice medium). Check the value field of the event for the voice event.	ADIEVN_CP_VOICE, with ADIEVN_CP_VOICE_MEDIUM in the value field
Remote party has spoken for a period longer than the second time threshold (voice long). Check the value field of the event for the voice event.	ADIEVN_CP_VOICE, with ADIEVN_CP_VOICE_LONG in the value field
Remote party has spoken for a period longer than the third time threshold (voice extended). Check the value field of the event for the voice event.	ADIEVN_CP_VOICE, with ADIEVN_CP_VOICE_EXTENDED in the value field
Remote party stopped speaking (voice end). Check the value field of the event for the voice event.	ADIEVN_CP_VOICE, with ADIEVN_CP_VOICE_END in the value field

When call progress terminates, ADIEVN_CP_DONE is generated.

System restrictions

When using the NOCC protocol, call progress functions can be run at any time. With all other protocols, call progress is under the control of the protocol until the call enters the connected state. Once the call is in the connected state, the application can run call progress functions.

Detecting tones

The tone detector runs a precise tone filter for a single or dual frequency tone. Each instance of the ADI service (for example, each context) has up to six programmable tone detectors. If the current telephony protocol employs an in-band clear-down tone detector, the first tone detector is not available. The tone detectors can generate the following events:

- ADIEVN_TONE_*n*_BEGIN
where *n* is the programmable tone ID (1-6)
- ADIEVN_TONE_*n*_END
where *n* is the programmable tone ID (1-6)

This topic presents:

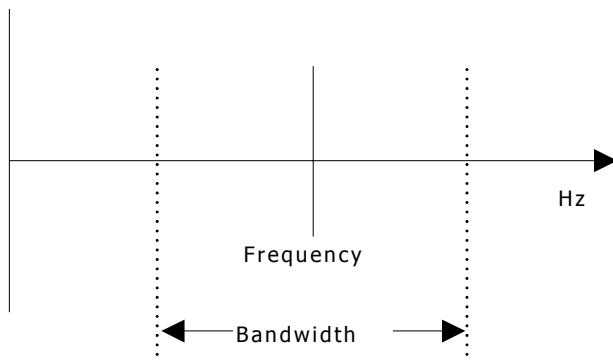
- Starting tone detection
- Stopping tone detection

Starting tone detection

In addition to the tone detector identifier that specifies a tone ID of 1 through 6, **adiStartToneDetector** takes four parameters that describe a single or dual frequency tone:

Parameter	Description
freq1	Frequency 1. The center frequency in Hz of a single tone or the first of two frequencies in a dual tone.
bandw	Bandwidth 1. The bandwidth around Frequency 1 that is acceptable.
freq2	Frequency 2. The center frequency in Hz of the second of two frequencies in a dual tone. Set this value to zero for single frequency tones.
bandw2	Bandwidth 2. The bandwidth around Frequency 2 that is acceptable. Set this value to zero for single frequency tones.

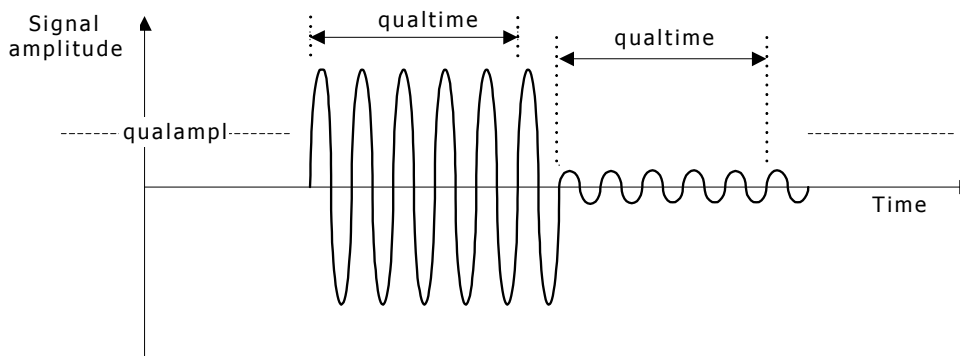
The range of frequencies detected is the center frequency plus or minus one-half of the bandwidth. The following illustration describes the tone detection frequency parameters:



You can further modify the tone detector's default behavior by specifying the following parameters (which reside in the ADI_TONEDETECT_PARMS structure) when invoking **adiStartToneDetector**, or by modifying the system defaults:

Parameter	Description
qualampl	Qualification amplitude; the broadband qualification level in dBm required to further qualify any energy as tone.
qualtime	Qualification time; the time in milliseconds in which the tone must be present before reporting ADIEVN_TONE_ <i>n</i> _BEGIN. After qualifying at tone, this parameter is used to qualify the absence of the tone to report ADIEVN_TONE_ <i>n</i> _END.

The following illustration describes the tone detection qualification parameters:



Note: Do not modify the `reflevel` and `reserved` parameters in `ADI_TONEDETECT_PARMS`. These parameters apply to the DSP algorithms and are provided for diagnostic purposes when working with NMS Technical Services.

Stopping tone detection

adiStopToneDetector immediately terminates a tone detector. The ADI service generates `ADIEVN_TONE_n_DETECT_DONE` with the value set to `CTA_REASON_STOPPED`.

The ADI service can also generate `ADIEVN_TONE_n_DETECT_DONE` with an error code, `ADIERR_XXX` or `CTAERR_XXX`, if the function is incorrectly started.

Generating tones

NMS Communications boards are capable of generating single and dual frequency tones.

Playing tones

adiStartTones enables the application to play a list of single or dual frequency tones. Each individual tone has the following attributes, which are stored in the ADI_TONE_PARMS structure:

Parameter	Description
ontime	A configurable active time period.
offtime	A configurable inactive time period following the active time period.
iterations	Number of times to repeat the tone.

The combined duration of ontime and offtime represents one complete cycle.

adiStartDTMF is a DTMF tone generator with programmable interdigit delays. The function accepts a string of digits and an ADI_DTMF_PARMS structure. The parameter structure allows you to specify interdigit pause duration for the comma and period characters, which can be interspersed with the DTMFs in the digit string.

Terminating tone generation

adiStopTones immediately terminates active tone generation. Regardless of which tone type is active, the ADI service generates ADIEVN_TONES_DONE with the value set to CTA_REASON_STOPPED.

The number of iterations is specified in ADI_TONE_PARMS, which is passed to **adiStartTones**. If the specified number of iterations is completed, the ADI service generates ADIEVN_TONES_DONE with the value set to CTA_REASON_FINISHED.

If an error occurs in starting the function, the DONE event is sent with the value set to ADIERR_xxx or CTAERR_xxx.

System restrictions

Because only one function can drive the output of the telephone line, the following functions are mutually exclusive:

- Tone generation
- Voice playback
- Voice record when beep is enabled

Collecting digits

The ADI service provides both synchronous and asynchronous digit collection functions. Call control must be in the connected state to activate the digit collection functions and the application must leave the DTMF detector enabled. DTMF detection parameters are loaded when the protocol is started.

In general, digit collection operates as follows:

- When a caller depresses a digit on the handset, the board sends ADIEVN_DIGIT_BEGIN to the application, and the digit becomes available.
- When the caller releases the key, the board sends ADIEVN_DIGIT_END to the application.
- Each event's value field contains an ASCII value indicating the key pressed or released. The valid values are 0 through 9, * (asterisk), # (number sign), and A through D.

This topic presents:

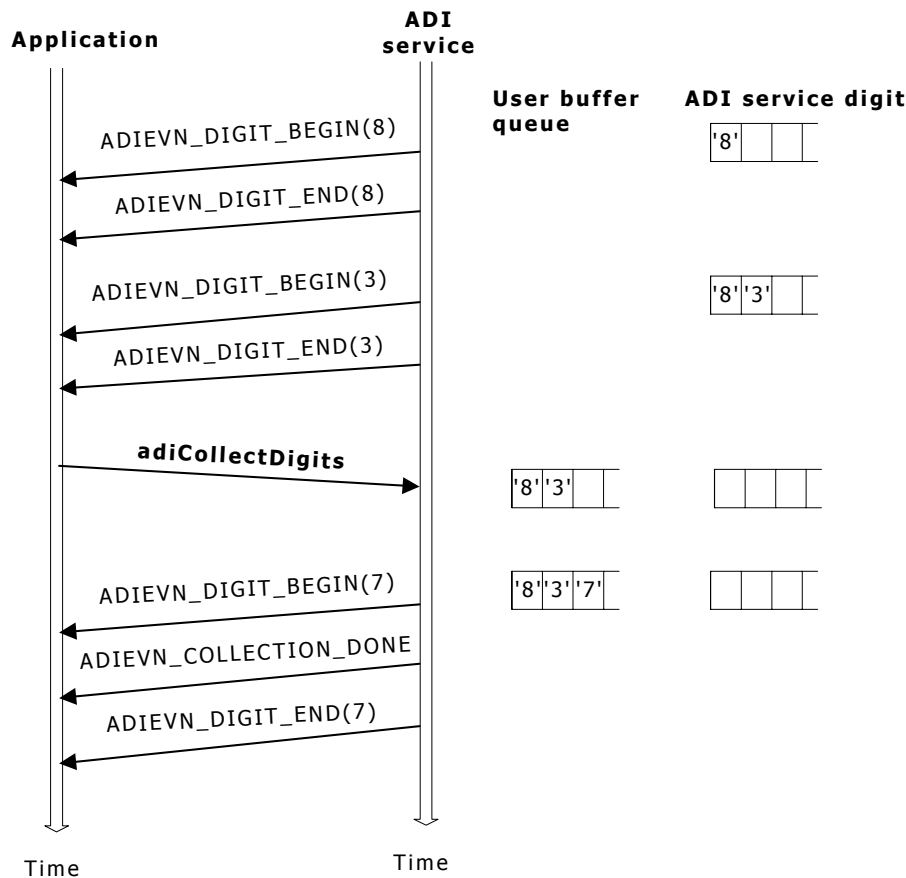
- Synchronous digit collection
- Asynchronous digit collection
- Modifying DTMF detection
- Terminating DTMF detection
- Improving DTMF using echo cancellation

Synchronous digit collection

The ADI service maintains an internal DTMF digit queue to store digits entered by the remote party. If the application is not actively collecting digits using **adiCollectDigits**, DTMFs entered by the remote party are appended to the queue, as shown in the following illustration.

The digit is stored in the ADI service digit queue when ADIEVN_DIGIT_BEGIN is received. If the ADI service digit queue is full when ADIEVN_DIGIT_BEGIN arrives, the oldest digit is discarded and the latest digit is queued. The ADI service digit queue can hold 64 digits.

The following illustration shows background digit collection:



The ADI service provides four synchronous functions that access the internal ADI service digit queue:

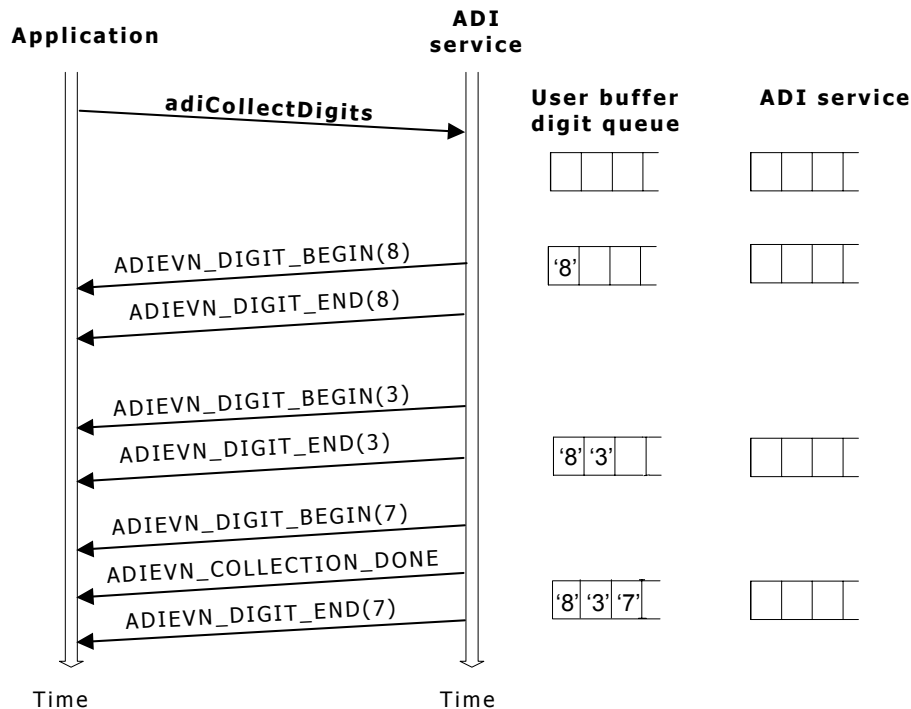
Function	Description
adiGetDigit	Retrieves a single digit from the ADI service internal digit queue, thus removing the oldest digit from the queue. If the queue is empty, a zero (0) is returned; otherwise, the ASCII value is returned.
adiInsertDigit	Inserts a digit at the end of the ADI service internal digit queue.
adiPeekDigit	Retrieves the oldest digit from the ADI service internal digit queue, without removing the digit from the queue.
adiFlushDigitQueue	Discards all digits stored in the internal digit queue.

None of these functions can be invoked if the application is actively collecting digits with **adiCollectDigits**.

Asynchronous digit collection

The ADI service enables applications to collect DTMF digit strings asynchronously into their own buffers. **adiCollectDigits** initiates digit collection into a user-specified buffer rather than into the ADI service digit queue. Digits are appended to the user-specified buffer until a terminating event occurs.

The following illustration represents asynchronous digit collection:



Because they perform read and write operations on the internal digit queue, an application cannot call **adiGetDigit** and **adiFlushDigitQueue** while actively collecting digits. The ADI service returns an error if an application attempts to modify the internal digit queue while digit collection is active.

You can modify the collection function's default behavior by redefining the parameters in **ADI_COLLECT_PARMS** when invoking **adiCollectDigits**.

Terminating asynchronous digit collection

The collection operation has programmable termination conditions. An application can also prematurely terminate the function by invoking **adiStopCollection**. In all cases, the ADI service sends **ADIEVN_COLLECTION_DONE** to the application, which indicates that collection finished. The value field contains the termination reason.

The **maxdigits** argument in **adiCollectDigits** specifies the maximum number of digits to collect. Only digits written to the user buffer are counted. For example, digits discarded because they are not in the acceptable list are not counted. Digit collection terminates when this maximum digit count is reached.

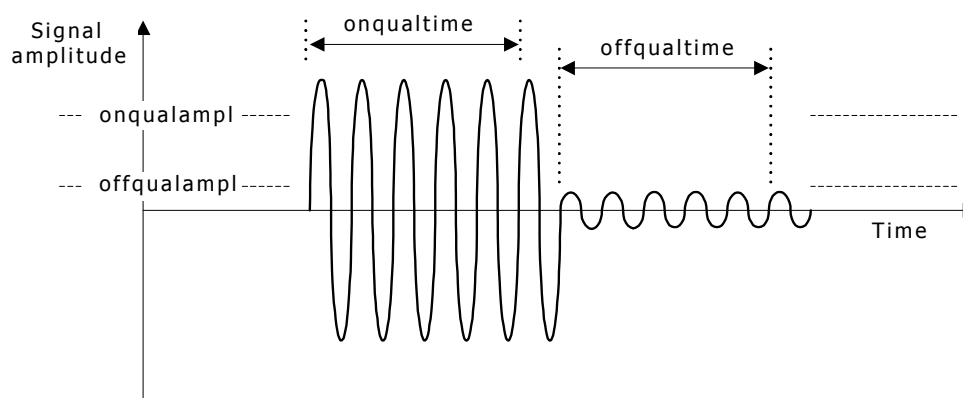
If digits are stored in the ADI service's internal digit queue when **adiCollectDigits** is invoked, the ADI service processes the digits individually from the front of the ADI service digit queue. Each digit processed is checked against a list of acceptable digits, and a list of terminating digits, which are stored in the validDTMFs and terminator fields in the ADI_COLLECT_PARMS structure. Invalid digits are discarded. Terminating digits cause digit collection to terminate.

When digit collection terminates, ADIEVN_COLLECTION_DONE is delivered to the application. For information on reasons, see **adiCollectDigits**.

Digit collection can also terminate when the interdigit timeout value in ADI_COLLECT_PARMS is exceeded.

Modifying DTMF detection

You can modify the DTMF detector's default behavior when invoking **adiStartProtocol** or **adiStartDTMFDetector**. The DTMF detector parameters are stored in the ADI_DTMFDETECT_PARMS structure. The following illustration shows the effect of these parameters:



Note: Do not modify onthreshold and offthreshold. These parameters apply to the DSP algorithms and are provided for diagnostic purposes when working with NMS Technical Services.

Terminating DTMF detection

adiStopDTMFDetector immediately terminates DTMF detection. The ADI service generates ADIEVN_DTMFDETECT_DONE with the value set to CTA_REASON_STOPPED.

The ADI service can also generate ADIEVN_DTMFDETECT_DONE with an error code, ADIERR_XXX or CTAERR_XXX, if the function is incorrectly started.

Improving DTMF using echo cancellation

Echo cancellation improves the ability of the DTMF detector to recognize digits during play, a capability referred to as DTMF cut-through performance.

Using echo cancellation with the DTMF detector allows the use of a more selective DTMF detector, which improves resistance to talk-off (the false detection of digits in a speaker's voice).

AG and CG boards have an alternate DSP file (*dtmfe*) used specifically with echo cancellation. To load *dtmfe*:

Step	Action
1	Locate the reference to <i>dtmf.xxx</i> in the DSP.C5x[<i>x</i>].Files[<i>y</i>] keyword (DSP.C5x[<i>x</i>].Files for CG boards), where <i>xxx</i> is either <i>.dsp</i> , <i>.m54</i> , or <i>.f54</i> . There may be no file extension.
2	Change <i>dtmf</i> to <i>dtmfe</i> .
3	Save the changes and re-initialize the board.

Echo cancellers of moderate length and adaptation time typically provide improvement of 10 to 15 dB in DTMF cut-through performance.

Controlling echo

NMS Communications echo cancellers can be used to implement echo control for the following applications:

Application	Implementation
PSTN terminal	Improves DTMF detection (DTMF cut-through) or automatic speech recognition performance by eliminating leakage of playback audio into the receive signal path. This behavior typically applies to IVR or voice mail applications of NMS Communications boards.
Network echo control	Eliminates talker echo so that peer-to-peer human communications do not suffer the annoying effects. This behavior typically applies to IP telephony gateway applications on NMS Communications boards.

This topic presents:

- Echo cancellation examples
- Echo canceller components
- Specifying echo canceller parameters
- Configuring boards for echo cancellation
- Recommendations for controlling echo

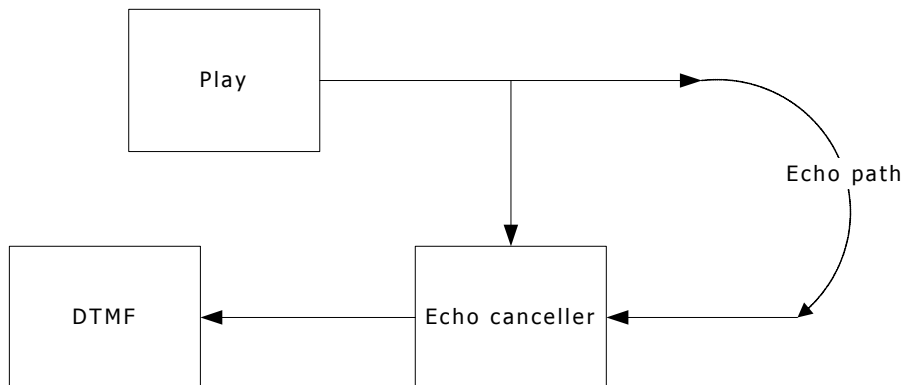
Echo cancellation examples

DTMF cut-through example

In an IVR application, the user typically uses DTMF keys to make option selections. Since the user calling into the IVR system does not always wait for the whole message to be played, an echo canceller is needed to cancel the local and near-end echo of the prompt played by the application. Canceling the echo enables the local DTMF detector to recognize a received tone during the time the message is played.

The echo canceller improves the signal-to-noise ratio as seen by the DTMF detector on the NMS Communications board. The useful signal is the received DTMF signal and the noise is the echo of the message prompt played by the board.

Similar to the example of DTMF cut-through, the echo canceller also helps in improving cleardown tone detection. The following illustration shows echo cancellation for DTMF cut-through:



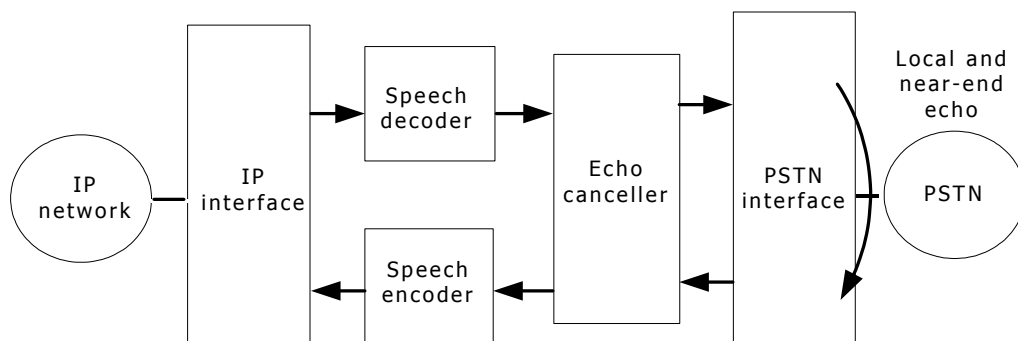
Host-based ASR example

This application is similar to the DTMF cut-through example with the automatic speech recognition (ASR) system replacing or augmenting the DTMF detector for control of the IVR session. ASR algorithms require a high performance echo canceller. A prompt is played out on the board. The user commands the application by saying a keyword (for example, a number or a name) to make a selection. The person's response is processed by the software that runs on the host or on the board. A necessary condition for a correct recognition is an echo-free received signal. The echo canceller on the local board must respond quickly to any changes and totally cancel the echo without distorting the incoming signal.

The echo canceller provides settings to optimize performance for ASR applications. The ASR application may need to defeat its endpointing until the echo canceller has fully converged. Empirical tests have shown that the echo suppressor part of the echo canceller (sometimes called the non-linear processor or NLP) must be disabled. These controls can be used through ADI functions.

IP telephony gateway (network echo canceller) example

Another important application of echo cancellation can be found in IP telephony gateways. The following illustration shows the two gateways:



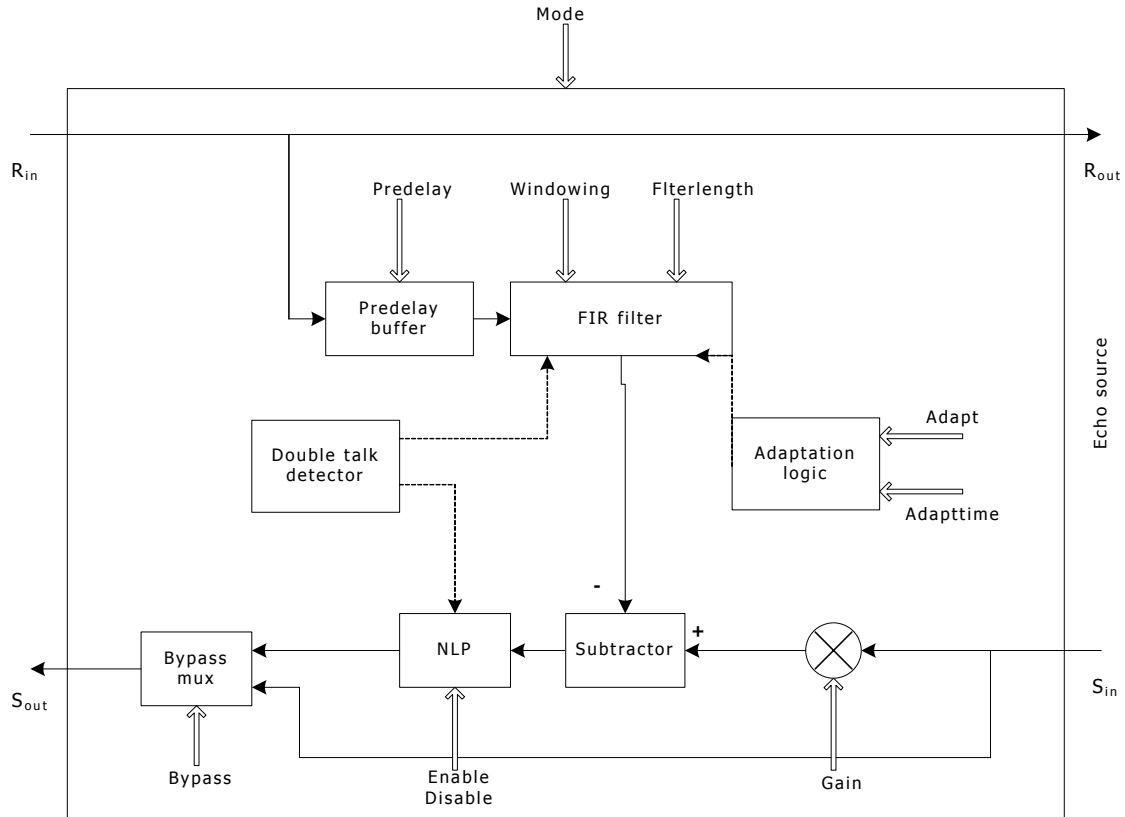
For a two-wire connection, the gateway echo canceller cancels the local echo generated by the on-board hybrid and the near-end echo generated by the near-end hybrid. The returned echo level for the echo canceller must be as low as possible because the one way delay for this type of connection can be 100 ms or more.

For a four-wire connection to the PSTN, the echo canceller cancels the near-end echo, or in some cases, no echo at all. For a near-end echo, the requirements are the same as in the previous case.

Without proper echo control in an IP telephony application, annoying echo can be heard by both speakers in a duplex voice conversation. The longer the delay through the IP network, the more unpleasant the effects of any residual or uncanceled echo.

Echo canceller components

The following illustration shows the structure of the echo canceller:



Echo cancellers are four port devices, two ports facing the near end and two ports facing the far end. The four ports are: R_{in} , R_{out} , S_{in} , and S_{out} .

R stands for receive if the port is situated in the receive path. S stands for send if the port is situated in the send path. The subscripts _{in} and _{out} define the input and output ports of the echo canceller on the corresponding path.

The main components of the echo canceller are:

Component	Description
Predelay buffer	Signals sent into the echo path enter a predelay buffer prior to being operated on by the FIR filter. Depending on the board, the predelay can be set from 0 to a maximum of 20 ms delay. This predelay is introduced to compensate for the pure delay in the echo path.
Finite impulse response (FIR) filter	The echo canceller FIR filter tries to mimic the echo path. The coefficients or taps of the FIR filter determined by the adaptation logic, determine the FIR filter response. The FIR filter converges to mimic the echo channel when the coefficients of this filter equal the impulse response of the echo path. The length of the FIR filter determines how much of an echo is covered by the echo canceller. The FIR filter and the adaptation logic can be referred to as an adaptive filter.
Subtractor	The subtractor subtracts the output of the FIR filter from the signal in the send path. If the adaptation performs well (for example, the echo path has been exactly identified), S_{in} is equal to the adaptive filter output (echo estimate) and the difference is zero. Because the adaptive filter can never match the echo path exactly, the difference between S_{in} and the echo estimate is never zero. This difference is called the error signal and is used by the adaptive filter to improve its performance. The better the estimation of the echo path, the smaller the energy of the error signal. The attenuation of the signal at the output of the subtractor in relation to the S_{in} signal is denoted as echo returned loss enhancement (ERLE).
Adaptation logic	The adaptation logic updates the FIR filter coefficients using the error signal. A modified least mean square (LMS) algorithm is used to modify the coefficients in an iterative fashion. The application can freeze or stop this adaptation, or reset the value of the coefficients to restart convergence.
Double-talk detector	The double-talk detector detects when both callers speak at the same time (IP telephony application) or when DTMF is input to the system at the same time as audio playback (IVR). In the presence of double-talk, this detector sends a command to the adaptation logic to stop or slow the adaptation of the coefficients. Detecting the double talk situation is critical for correct operation of the echo canceller. If adaptation continues during double-talk, the adaptive filter modifies its coefficients based on the information contained in the S_{in} signal. In this case, this is the sum of the echo of R_{out} signal and the signal produced by the near-end talker. The adaptation would therefore be erroneous.
Non-linear processor (echo suppressor)	The non-linear processor is a device with a defined suppression threshold level in which signals having a level detected: <ul style="list-style-type: none"> Below the threshold are suppressed. Above the threshold are passed (although the signal can be distorted). The non-linear processor functions only during single talk situations. The non-linear processor attenuates the residual echo that could not be cancelled by the adaptive filter.
Input gain	For all boards except QX boards, the application can provide input signal gain or loss.
Bypass	The application can bypass the echo canceller and restart at any time. Use Bypass when voiceband modems or fax machines terminate both ends of the connection in an IP telephony application.

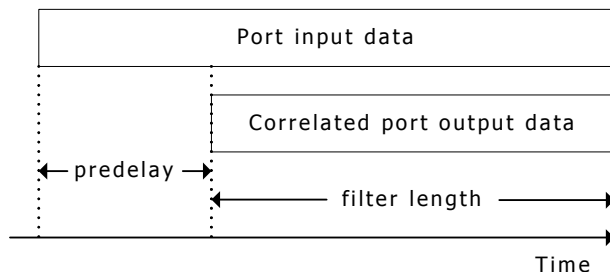
Specifying echo canceller parameters

If you use echo cancellation in your application, you may need to modify the `callctl.mediamask` in `ADI_START_PARMs` (or the `mediamask` in `NCC_ADI_START`) before you start a telephony protocol. The `mediamask` controls which functions are running or reserved when the call enters the connected state. Reserved indicates that the DSP MIPS have been committed to the operation before the operation starts. The application must reserve DSP resources in advance by using `mediamask` for DTMF detection, silence detection, cleardown detection, and echo cancellation.

The ADI service initiates echo cancellation when a telephony protocol is started. The appropriate parameters must be set before calling **`adiStartProtocol`** or **`nccStartProtocol`**. For information on the echo cancellation parameters, refer to `ADI_START_PARMs` on page 264.

The echo canceller parameters can be modified after the echo canceller is started by calling **`adiModifyEchoCanceller`**.

For all board types, the `predelay` parameter time shifts the correlation buffer. This enables shorter filter lengths to be shifted in time, allowing more echo energy to be captured, as shown in the following illustration:



The default mode (`mode = 1`) chooses the best possible echo cancellation for the available DSP power on the board. Choosing echo cancellation parameters that consume more DSP power than is available can result in errors when all ports are active. To determine whether your boards support echo cancellation, refer to *Default filter length and adaptation time values* on page 62.

Configuring boards for echo cancellation

Echo cancellation requires board-specific settings.

Note: The PacketMedia HMP process does not support echo cancellation.

Configuring AG boards for echo cancellation

For AG boards, configure the system for echo cancellation by editing the board keyword file. Add `echo.m54`, `echo_v3.m54`, or `echo_v4.m54`, depending on the features you require, to the list of files in `DSP.C5x[x].Files[y]`.

For information on DSP file features, see *DSP file summary* on page 269.

To enable echo cancellation with the board's default settings, set the parameter `ADI.START.echocancel.mode` or `NCC.X.ADI_START.echocancel.mode` to 1. The AG 4000, AG 4000C, AG 4040, and AG 4040C boards do not have default settings. See *Default filter length and adaptation time values* on page 62.

For AG boards, as the predelay value is in increments, the correlated data buffer is shifted later in time. The predelay can be adjusted to center the correlated data on most of the echo energy. The valid range is from 0 to 20 milliseconds.

Refer to the board's installation and developer's manual for more information.

Configuring QX boards for echo cancellation

For QX boards using analog ports, predelay must be set to 0 milliseconds to capture local and near-end echo. QX boards reduce the level of the echo less than -60 dBm. Therefore, depending on the level of the transmitted signal and impedance adaptation, the improvement of the QX boards can be greater than -30 dB.

Refer to the *QX 2000 Installation and Developer's Manual* for more information.

Configuring CG boards for echo cancellation

The resource definition string and the list of data processing modules (DPM) loaded on the DSPs on the CG boards have a default setup that includes echo.

To configure a CG board for echo cancellation, edit the board keyword file. Add *echo.f54*, *echo_v3.f54*, or *echo_v4.f54*, depending on the features you require, to the list of files in DSP.C5x[**x**].Files. For information on DSP file features, see *DSP file summary* on page 269.

CG 6565/C boards and CG 6060/C boards use C5441 DSPs and not C5420 DSPs for applications. The DSP files have *.f41* extensions instead of *.f54* extensions. For information about configuring hardware echo cancellation on CG 6565/C boards and CG 6060/C boards, refer to the board installation and developer's manual.

The default echo, Echo.In20_apt25 specified in the resource definition string, has a 20 ms filter length and an adapt rate of 25 percent of the maximum adaptation rate. If an echo different from Echo.In20_apt25 is needed, change the resource definition string. Replace the current echo in the resource definition string with the new echo.

Note: Changing a function in the resource definition string can decrease the number of ports that run on the board. Each DSP function has its own resource requirement. If the new function has higher resource requirements than the function it is replacing, the number of ports the board can run can be less.

Refer to the board installation and developer's manual for more information.

Default filter length and adaptation time values

To enable echo cancellation with default settings, set ADI.START.echocancel.mode or NCC.X.ADI_START.echocancel.mode to 1.

The following table shows the default filter length and adaptation time values for each board type:

Board type	Filter length	Adaptation time
CG	20 ms	200 (25 percent of maximum adaptation rate)
AG 2000, AG 2000C	4 ms	100 ms
AG 4000, AG 4000C, AG 4040, AG 4040C	0 ms	0 ms
QX 2000	20 ms	Not used.

To enable echo cancellation with specific parameters:

- Set ADI.START.echocancel.mode or NCC.X.ADI_START.echocancel.mode to 2.
- Set ADI.START.echocancel.filterlength or NCC.X.ADI_START.echocancel.filterlength to values of your choosing.
- Set ADI.START.echocancel.adapttime or NCC.X.ADI_START.echocancel.adapttime to values of your choosing.

Features

The following table provides general information about the echo canceller features:

Features	AG 2000, AG 2000C, AG 4000, AG 4000C, AG 4040, AG 4040C	CG	QX 2000
Filter length	2,4,6,8,10,16,20,24,32, 40,48,64 ms	2,4,6,8,10,16,20,24, 32,40,48,64 ms	1,2...20 ms
Echo pre-delay	0,1,2...20 ms	0,1,2...20 ms	0,1,2...20 ms
Double talk detector	Yes	Yes	Yes
Input gain	Yes	Yes	No
Echo suppressor enable/disable	Yes	Yes	Yes
Adaptation enable/disable	Yes	Yes	Yes
Windowing enable	No	No	Yes
Bypass	Yes	Yes	Yes
Comfort noise generation	Yes	Yes	No
Tone disabling	Yes	Yes	No

Performance parameters

The following table provides general information about the echo canceller performance parameters:

Performance	AG 2000, AG 2000C, AG 4000, AG 4000C, AG 4040, AG 4040C	CG	QX 2000
Minimum echo return loss (ERL _{min}). For all values of ERL greater than ERL _{min} , the echo canceller delivers the expected performance. If the real ERL is less than the ERL _{min} , the echo canceller does not function correctly.	6 dB	6 dB	6 dB
Maximum echo return loss enhancement (ERLE).	33 dB	33 dB	33 dB
Non-linear processor loss. An additional loss introduced in the reception path, only when pure echo is received (no near-end speech).	36 dB	36 dB	12 dB, 24 dB
Typical convergence time on speech. Convergence time depends on the transmitted signal, double talk events, and adaptation time parameter for AG 2000, AG 2000C, AG 4000, AG 4000C, AG 4040, AG 4040C, and CG boards and on echo return loss of the network. The convergence time can be greater than the values presented in this table.	Less than 1 second. Obtained using <i>echo_v3.x54</i> . For <i>echo.x54</i> , the typical convergence time on speech is < 4 seconds.	Less than 1 second. Obtained using <i>echo_v3.x54</i> .	Less than 1 second.

Recommendations for controlling echo

Transmission level planning and echo

For IP telephony applications, proper audio levels and echo are tightly coupled. It is desirable to provide adequate listening levels; but increasing system gains anywhere in the four-wire trunk portion of a connection can make proper echo control difficult to attain under a wide range of telephony equipment and connection scenarios. In general, have no more than a zero dB of gain in each direction of the complete four-wire part of a connection. If it is necessary to increase the gain prior to a low-bit rate codec for example, there should be a commensurate loss at the output of the decoder.

Using Microsoft NetMeeting or other IP telephony clients

In IP telephony applications, the connection can be asymmetric. For example, you can talk on a telephone through an IP telephony gateway connected through an IP link to someone using Microsoft NetMeeting client on the remote end. At this NetMeeting client, the microphone and loudspeakers should not be used; a microphone headset is preferred. With a microphone and speaker combination at the NetMeeting client, the person on the telephone end of the connection will hear considerable echo due to the acoustic, loudspeaker-to-microphone acoustic coupling.

Delay and echo

In IP telephony applications, a user's tolerance to echo in a telephone conversation is reduced by the more end-to-end delay there is in the connection. IP packet delay is caused by routers and WAN facilities. High packet inter-arrival packet jitter usually must be absorbed by jitter buffers in the media gateway. The more jitter there is in the IP network, the longer the jitter buffer must be so the user does not experience poor audio quality due to packet loss.

Designers of IP telephony applications must reduce the number of routers and the amount of packet jitter so that any residual, uncanceled echo does not unnecessarily degrade the quality of the telephone connection.

Non-voice terminals (FAX and modem pass-through)

For IP telephony applications, it is desirable to handle non-voice communication devices such as modems. Modem transport can be handled by setting up a full duplex G.711 MSPP channel. Disable the echo canceller since it impairs both FDM (frequency division multiplexing) and EC (echo cancelling) modem transmission.

For T.30 FAX, T.38 can be used as a packet transport, or the MSPP channel can be set to G.711. In either case, disable the echo canceller.

Automatic speech recognition

Speech control of an IVR application can present special echo control challenges. Consider the following recommendations to improve the ability of the speaker to cut through a voice prompt to control the application:

- The echo suppressor should be disabled.
- The echo canceller with the fastest adaptation time should be used.

Depending on which board you use, you may be able to select an echo canceller that has faster convergence (reduces echoes more quickly). For example, the CG 6000C echo canceller can be configured for a 100 percent adaptation rate (fast). With a 20 ms echo coverage, this canceller requires 5.40 MIPS.

Minimization of two-wire switching

Hybrids in telephony circuits convert two-wire transmission to and from four-wire transmission. Most modern circuit switched telephony switching is done at four-wire points in a connection. Older two-wire switching still exists. Each interface from a two-wire to a four-wire connection can be a source of echoes. Therefore, wherever possible, minimize the use of two-wire switching.

Detecting energy

The ADI service is capable of running an energy detector that examines the in-band signal and reports energy and silence transitions.

Note: Do not use the energy detector if you are using voice activity detection.

This topic presents:

- Starting energy detection
- Stopping energy detection

Starting energy detection

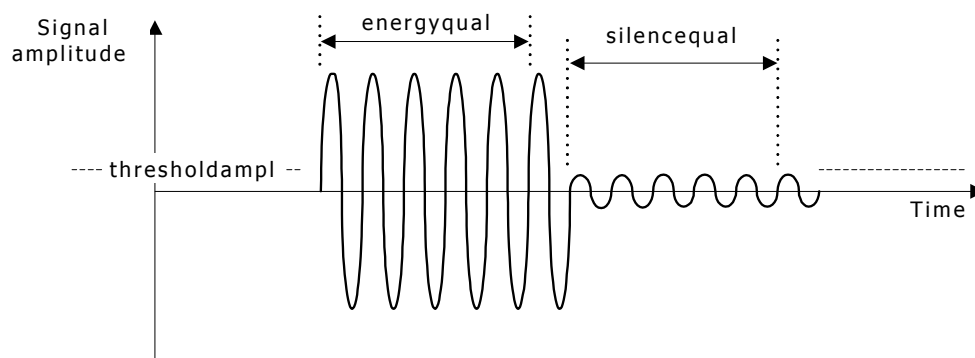
adiStartEnergyDetector takes the following qualification time parameters:

Parameter	Description
energyqual	Energy qualification time; the time, in milliseconds, that energy must be present to report an energy event.
silencequal	Silence qualification time; the time, in milliseconds, that energy must be absent to report a silence event.

You can modify the energy detector's default behavior by specifying the following parameters (stored in ADI_ENERGY_PARMS) when invoking **adiStartEnergyDetector**:

Parameter	Description
thresholdampl	Threshold amplitude; the noise level, in dBm, below which is considered silence. Noise above this level is further qualified as energy.
deglitch	Deglitch time; the minimum time, in milliseconds, before a transition between energy and silence is recognized.
autostop	Automatic stop; controls whether the energy detector continues running after the first event. By default, automatic stop is enabled.

The following illustration shows the energy detection parameters:



The energy detector generates the following events:

- ADIEVN_SILENCE_DETECTED
- ADIEVN_ENERGY_DETECTED
- ADIEVN_ENERGY_DETECT_DONE

Stopping energy detection

adiStopEnergyDetector immediately terminates energy and silence detection. The ADI service generates ADIEVN_ENERGYDETECT_DONE with the value set to CTA_REASON_STOPPED, which means that the operation was stopped by application request.

The ADI service also generates ADIEVN_ENERGYDETECT_DONE, and ADIERR_***, CTAERR_***, or CTA_REASON_FINISHED. When set to autostop (one-shot), the value is set to CTA_REASON_FINISHED and the size field contains either ADIEVN_ENERGY_DETECTED or ADIEVN_SILENCE_DETECTED.

Detecting voice activity

NMS Communications AG and CG boards, as well as the PacketMedia HMP process, provide a voice activity detector that suppresses user voice silence during dialogues with a voice recognition system. By preventing silent data from being sent to the application for ASR processing, host processing resources can be conserved.

The voice activity detector provides the following features:

Feature	Description
Voice activity detection	Detects audio energy and triggers data transmission only when speech is present.
Pre-speech buffering	When the voice activity detector detects speech, the board runtime immediately sends the previously filled buffer to the host, reducing the problem of clipped speech.
Voice event signaling	Sends SPEECH_BEGIN and SPEECH_END messages, and noise and signal energy to the host application.
Recorded stream control	Pauses and resumes sending recorded data to the runtime, while keeping the voice activity detector algorithm active on the DSP.

The voice activity detector enables host application control of voice activity detection features, including:

- Start (with new or default parameters) and stop voice activity detection.
- Update voice activity detection parameters on the fly.
- Enable and disable voice activity detection signaling.
- Pause and resume the recorded stream from the board to the host.

The voice activity detector has a fixed delta threshold that allows it to adapt to the background noise level. When the voice level is higher than the background noise level by a specified delta, the detector sends a SPEECH_BEGIN event to the application. When the voice level falls below the background noise level, the detector sends a SPEECH_END event.

Use the voice activity detector with any ASR application that is recording with one of the following encoding formats:

- ADI_ENCODE_MULAW
- ADI_ENCODE_ALAW
- ADI_ENCODE_PCM8M16

Voice activity detection does not interfere with other existing capabilities such as DTMF detection and echo cancellation.

Configuring boards for voice detection

To configure the system for voice activity detection, edit the board keyword file as follows:

For these boards...	Add this DSP file...	To this keyword...
AG	<i>rvoice_vad.m54</i>	DSP.C5x[x].Files[y]
CG	<i>rvoice_vad.f54</i>	DSP.C5x[x].Files

For example:

```
DSP.C5x[1..31].Files = dtmf rvoice_vad
```

To configure the PacketMedia HMP process for voice activity detection, set the value of the EnableVAD keyword to Yes. For example:

```
EnableVAD = Yes
```

Configure dynamic buffer allocation on the board to prevent host underruns.

You can also configure CG boards for voice activity detection by defining a DSP resource pool and specifying *rvoice_vad* as the resource definition:

```
Resource[0].Definitions = (dtmf.det_all & rvoice_vad.rec_alaw & rvoice_vad.rec.play_alaw)
```

Note: QX boards do not support voice activity detection.

Using voice activity detection

Voice activity detection and voice activity detection messaging are disabled by default. To enable voice activity detection, call **adiCommandRecord** on an actively running ADI recording function (such as **adiRecordAsync**).

ADIEVN_RECORD_STARTED must be received before calling **adiCommandRecord**.

You can perform the following functions using **adiCommandRecord**:

- Enable and disable voice activity detection
- Configure voice activity detection with application parameters
- Enable and disable voice activity detection messaging
- Pause and resume voice streaming from the board to the host

Sending and receiving FSK data

The ADI service is capable of sending and receiving frequency shift key (FSK) data. The transmit function is an implementation of the modem portion of Bellcore advisory TA-NWT-000030. The basic modulation is continuous-phase binary FSK at 1200 baud between 1200 Hz (mark) and 2200 Hz (space). The only supported baud rate is 1200.

This topic presents:

- Sending FSK data
- Terminating FSK data transmission
- Receiving FSK data
- Terminating FSK data reception

Sending FSK data

To send FSK data, call **adiStartSendingFSK**.

While it is running, the FSK transmitter is in one of the following states:

- Idle
- Sending channel seizure
- Sending mark
- Sending data

An FSK transmission consists of a sequence of states used to transmit a data packet, and it is bracketed by silence or the idle state.

An FSK transmission with analog display services interface (ADSI) feature phones consists of the following sequence of data: sending mark, sending data. The sending channel seizure state never occurs in ADSI feature phones. Therefore, `noseizureflag` in the `ADI_FSKSEND_PARMS` structure must be set to 1.

Parameters for sending FSK data

adiStartSendingFSK requires the following parameters defined in the `ADI_FSKSEND_PARMS` structure:

Field name	Description	Default value	Valid values
<code>noseizureflag</code>	0: Allows send channel seizure state and sending mark state. 1: Allows only sending mark state.	1	0, 1
<code>level</code>	Transmit output level.	-14 dBm	-
<code>seizetime</code>	Length of channel seizure in milliseconds; ignored if <code>noseizureflag</code> is set to 1.	1000	-
<code>marktime</code>	Length of initial mark signal in milliseconds.	500	-
<code>baudrate</code>	Transmission baud rate.	1200	1200

Terminating FSK data transmission

Use **adiStopSendingFSK** to stop the send function. The event value field contains CTA_REASON_STOPPED. The number of bytes sent before the function was stopped cannot be determined.

Receiving FSK data

Call **adiStartReceivingFSK** to enable an application to receive FSK data. While it is running, the FSK receiver is in one of the following four states:

- Idle
- Receiving channel seizure
- Receiving mark
- Receiving data

A complete packet of FSK data consists of either of the following sequences:

- Receive channel seizure, receive mark, receive data
- Receive mark, receive data

During the receive process, errors can occur that cause the function to terminate prematurely. If errors occur, ADIEVN_FSK_RECEIVE_DONE is sent with one of the following reasons:

Reason	Description
ADI_REASON_DROP_IN_DATA	Stopped because of drop in data. The noseizureflag signal dropped to silence during data, not during stop mark period.
ADI_REASON_BAD_STOP_BIT	Stopped because of data framing error. The stop bit at the end of data was a space, not a mark.

Parameters for receiving FSK data

adiStartReceivingFSK requires the following parameters defined in the ADI_FSKRECEIVE_PARMS structure:

Field	Description	Default value
minlevel	Required minimum receive level.	-35 dBm
minmark	Required minimum initial mark and seizure time, in milliseconds.	10
mindrop	Minimum dropout to silence before a packet is considered terminated, in milliseconds.	5
baudrate	Transmission baud rate.	1200

Terminating FSK data reception

Use **adiStopReceivingFSK** to stop the receive function. The event value field contains CTA_REASON_STOPPED. The size of the partial buffer received is stored in the size field of the event structure.

Performing low-level call control

The ADI service includes functions that enable applications to perform call control from the host. These functions are typically used with the NOCC (no call control) protocol. Their use is restricted when running other CAS protocols.

For example, all channel associated signaling (CAS) protocols reserve out-of-band signaling, so ADI service functions that perform out-of-band signaling may not be available. For information about CAS protocols, refer to the *NMS CAS for Natural Call Control Developer's Manual*.

The following low-level control functions are available in the ADI service:

If you want to...	Then use...
Assert an out-of-band signaling pattern	adiAssertSignal
Pulse an out-of-band signaling pattern for a duration	adiStartPulse
Start detection of out-of-band signaling bits	adiStartSignalDetector
Stop detection of out-of-band signaling bits	adiStopSignalDetector
Query the current inbound out-of-band signaling bits	adiQuerySignalState
Start DTMF/pulse dialing of digits	adiStartDial
Stop DTMF/pulse dialing of digits	adiStopDial

The out-of-band signaling functions relate to either the physical out-of-band signal bits of digital protocols or the control of analog interface boards. In both cases, four signaling bits are addressed: A, B, C, and D, often written as ABCD, and defined by a bit mask (0x8, 0x4, 0x2, and 0x1, respectively). When using these functions, refer to the appropriate manual for your telephone line interface board.

adiStartDial and **adiStopDial** enable you to perform dialing operations when you are not running formal call control.

Using on-board timers

The ADI service supports one on-board timer per context on a board. This on-board timer has 10 ms resolution. The timer generates periodic events. You specify both the period and number of events when invoking **adiStartTimer**.

Starting the timer

To start the timer, call **adiStartTimer** and pass a context, a timeout value, and an event count value. A DONE event is generated when the timer expires.

If the count value is greater than one, a tick event is generated for each expiration of the timeout with a DONE event for the final expiration.

The timer can be reset or restarted with an additional call to **adiStartTimer**. When restarted, previous timer definitions are discarded and the timer begins with the new parameters.

Note: Unlike most Natural Access asynchronous functions, the timer is not automatically stopped when a call is released.

Start timer events

The following table lists the start timer events:

Event	Description
ADIEVN_TIMER_DONE	Once the timer has completed (expired), the ADI service generates a DONE event with the value field set to CTA_REASON_FINISHED. A DONE event is received with an error in the value field only if the board has an error. If the timer is stopped by calling adiStopTimer , the value field is CTA_REASON_STOPPED.
ADIEVN_TIMER_TICK	If count is greater than 1, the ADI service generates a tick event for the first count-1 expirations.

Stopping the timer

adiStopTimer stops the timer started with **adiStartTimer**. Once the timer has stopped, a DONE event is generated.

Stop timer event

The following table lists the stop timer event:

Event	Description
ADIEVN_TIMER_DONE	When the timer is stopped, a DONE event is generated with the value field set to CTA_REASON_STOPPED.

4

Function summary

Telephony protocol functions

After setting up the ADI service, you must start a telephony protocol on each context to perform telephony activities. The NCC service provides a null protocol, NOCC, for applications that do not require call control. Refer to the *NMS CAS for Natural Call Control Developer's Manual* for a list of telephony protocols and parameters and for information about controlling calls under specific trunk control protocols (TCPs).

Record and play functions

The ADI service provides functions for recording and playing speech data.

Initiating record and play operations

Choose a set of functions to initiate record and play operations as appropriate to your application's data transfer method, according to the following table:

Operation	Memory transaction	Asynchronous	Callback
Play	adiPlayFromMemory	adiPlayAsync	adiStartPlaying
Record	adiRecordToMemory	adiRecordAsync	adiStartRecording

Terminating record and play operations

The ADI service provides the following functions to stop record and play regardless of the data transfer method:

Function	Synchronous/ Asynchronous	Description
adiStopPlaying	Asynchronous	Terminates playing.
adiStopRecording	Asynchronous	Terminates recording.

Using buffer management functions

For the asynchronous data transfer methods, a buffer is submitted using one of the following functions:

Function	Synchronous/ Asynchronous	Description
adiSubmitPlayBuffer	Asynchronous	Supplies a buffer to an asynchronous play operation.
adiSubmitRecordBuffer	Asynchronous	Supplies a buffer for an asynchronous record operation.

Using status and modification functions

The following functions provide status information or modify an active record or play operation:

Function	Synchronous/ Asynchronous	Description
adiModifyPlayGain	Synchronous	Changes the gain applied to the speech while playing.
adiModifyPlaySpeed	Synchronous	Changes the play speed while playing.
adiGetPlayStatus	Synchronous	Retrieves play (or last play) status.
adiGetRecordStatus	Synchronous	Retrieves record (or last record) status.
adiGetEncodingInfo	Synchronous	Returns frame size, data rate, and maximum buffer size for a given encoding format.
adiCommandRecord	Asynchronous	Sends a data array containing raw commands to an actively running recording function. Use this function to enable voice activity detection.

Call progress functions

Call progress analysis primarily allows the application to control and monitor the placement of outbound calls when not using call control (for example, when using the NOCC protocol). Call progress analysis can also be used at any time after a call is connected. For example, when receiving an inbound call, an application can start up a call progress analysis operation to detect modem tones, fax terminal tones, or voice.

The following ADI functions start and stop call progress analysis:

Function	Synchronous/ Asynchronous	Description
adiStartCallProgress	Asynchronous	Starts call progress analysis.
adiStopCallProgress	Asynchronous	Stops call progress analysis.

Tone detection functions

The following functions enable and disable detectors of precise tones, raw energy, and silence:

Function	Synchronous/ Asynchronous	Description
adiStartToneDetector	Asynchronous	Starts a precise tone detector.
adiStopToneDetector	Asynchronous	Stops a precise tone detector.
adiStartEnergyDetector	Asynchronous	Starts the energy detector (energy versus silence).
adiStopEnergyDetector	Asynchronous	Stops the energy detector (energy versus silence).

Tone generation functions

NMS boards generate single and dual frequency tones. The following ADI functions control tone generation:

Function	Synchronous/ Asynchronous	Description
adiStartTones	Asynchronous	Plays a sequence of user-defined tones.
adiStartDTMF	Asynchronous	Starts playing a sequence of DTMF tones; MF tones can also be generated.
adiStopTones	Asynchronous	Terminates playing tones.

Digit collection functions

The ADI service provides the following synchronous and asynchronous digit collection functions:

Function	Synchronous/ Asynchronous	Description
adiGetDigit	Synchronous	Retrieves a single digit from the internal digit collection queue.
adiInsertDigit	Synchronous	Inserts a digit at the end of the ADI service internal digit queue.
adiPeekDigit	Synchronous	Reads the first digit from the internal digit collection queue without removing it.
adiFlushDigitQueue	Synchronous	Flushes the internal digit collection queue.
adiCollectDigits	Asynchronous	Starts asynchronous digit collection.
adiStopCollection	Asynchronous	Terminates digit collection.

Echo cancellation functions

Echo cancellation improves DTMF/tone detection and speech recognition performance during playing by canceling any leakage of the playback audio from the receive signal path. It also improves peer-to-peer human communications in an IP telephony gateway application by eliminating talker echo.

Echo cancellation is configured as part of starting a protocol with **adiStartProtocol**. The echo canceller automatically starts when a call enters the conversation state.

The following functions modify echo cancellation parameters:

Function	Synchronous/ Asynchronous	Description
adiCommandEchoCanceller	Asynchronous	Sends commands to the echo canceller tone disabler.
adiModifyEchoCanceller	Asynchronous	Modifies the echo canceller parameters after echo cancellation is started.

DTMF and MF detection functions

The ADI service provides functions for enabling and disabling DTMF and MF detection. By default, DTMF detection is enabled when a call is established.

The following functions enable and disable DTMF and MF tone detection:

Function	Synchronous/ Asynchronous	Description
adiStartDTMFDetector	Asynchronous	Starts DTMF detection (default is on).
adiStopDTMFDetector	Asynchronous	Stops DTMF detection.
adiStartMFDetector	Asynchronous	Starts the MF tone detector.
adiStopMFDetector	Asynchronous	Stops the MF tone detector.

Frequency shift key data functions

NMS boards are capable of sending and receiving frequency shift key (FSK) data. The transmit function is an implementation of the modem portion of Bellcore advisory TA-NWT-000030. The basic modulation is continuous-phase binary FSK at 1200 baud between 1200 Hz (mark) and 2200 Hz (space). The only supported baud rate is 1200. Alternatively, the implementation is based on ITU V.23 FSK at 1200 baud between 1300 Hz (mark) and 2100 Hz (space).

The following functions are used to send or receive frequency shift key data:

Function	Synchronous/ Asynchronous	Description
adiStartSendingFSK	Asynchronous	Initiates sending frequency shift key data.
adiStopSendingFSK	Asynchronous	Stops the sending function.
adiStartReceivingFSK	Asynchronous	Receives frequency shift key data.
adiStopReceivingFSK	Asynchronous	Stops the receive function.

FSK modems are the low-level building blocks for analog display services interface (ADSI). The ADI service does not provide an ADSI programming interface, but you can use the FSK functions to implement ADSI applications.

Low-level call control functions

The following functions provide low-level access to the line interface. This access is typically needed only when using the NOCC (no call control) protocol. These functions are used when the application is directly controlling the line interface or when a CAS protocol is not needed. Use these functions carefully, especially if you are running a telephony protocol.

The following ADI functions provide low-level call control:

Function	Synchronous/ Asynchronous	Description
adiAssertSignal	Synchronous	Asserts an out-of-band signaling pattern. adiAssertSignal returns before the pattern is actually asserted.
adiStartPulse	Asynchronous	Pulses an out-of-band signaling pattern for a duration.
adiStartSignalDetector	Asynchronous	Starts detection of out-of-band signaling bits.
adiStopSignalDetector	Asynchronous	Stops detection of out-of-band signaling bits.
adiQuerySignalState	Asynchronous	Queries the current inbound out-of-band signaling bits.
adiStartDial	Asynchronous	Starts DTMF/MF/pulse dialing of digits.
adiStopDial	Asynchronous	Stops DTMF/MF/pulse dialing of digits.

On-board timer functions

The ADI service supports one application timer per port. This on-board timer has 10 ms resolution and can be used when the application is controlling the protocol from application space. The following timer functions are provided by the ADI service:

Function	Synchronous/ Asynchronous	Description
adiStartTimer	Asynchronous	Starts an on-board timer.
adiStopTimer	Asynchronous	Stops an on-board timer.

Configuration information functions

The following functions retrieve information about a system configuration or specific board. They also set the time or native play and record parameters:

Function	Synchronous/ Asynchronous	Description
adiGetBoardInfo	Synchronous	Retrieves information about the board.
adiGetBoardSlots adiGetBoardSlots32	Synchronous	Retrieves the board's MVIP configuration.
adiGetContextInfo	Synchronous	Retrieves the context status and configuration.
adiGetEEPromData	Synchronous	Retrieves OEM data for a given board.
adiGetTimeStamp	Synchronous	Converts an event timestamp to a count of the seconds elapsed since January 1, 1970.
adiSetBoardClock	Synchronous	Sets the time on an AG board, CG board, or PacketMedia HMP process.
adiSetNativeInfo	Synchronous	Sets parameters that enable applications to play and record media to and from RTP streams. In addition, this function enables applications to play media recorded from an RTP stream to a PSTN port and vice versa.

Board functions communicate only with the board's driver and not directly with the board. When opening the ADI service to use these functions, you do not specify an MVIP address. The `board_number` parameter in the `mvipaddr` structure can be set to `ADI_AG_DRIVER_ONLY`.

5

Function reference

Using the function reference

This section provides an alphabetical reference to the ADI service functions. A prototype of each function is shown with the function description, details of all arguments, and return values. Function information typically includes:

Supported board types	Each function supports one or more of the following board types: <ul style="list-style-type: none">• QX boards• AG boards• CG boards• PacketMedia HMP process
Prototype	The prototype is followed by a list of the function arguments. NMS Communications data types include: <ul style="list-style-type: none">• WORD (16-bit unsigned)• DWORD (32-bit unsigned)• INT16 (16-bit signed)• INT32 (32-bit signed)• BYTE (8-bit unsigned) If a function argument is a data structure, the complete data structure is defined.
Return values	The return value for a function is either SUCCESS or an error code. For asynchronous functions, a return value of SUCCESS indicates the function was initiated; subsequent events indicate the status of the operation. Refer to <i>Alphabetical error summary</i> on page 237 for a list of all errors returned by the ADI service functions.
Events	If events are listed, the function is asynchronous and is complete when the DONE event is returned. If no events are listed, the function is synchronous. Additional information such as reason codes and return values is provided in the value field of the event. Refer to <i>Alphabetical event summary</i> on page 243 for details of all ADI service events.
DSP file	Lists the DSP file that is required for this function. Refer to <i>DSP file summary</i> on page 269 for a list of DSP files.
Example	Example functions that start with Demo are excerpts taken from the demonstration code, which is shipped with the product. Example functions that start with my are excerpts taken from sample application programs. The notation <code>/* ... */</code> indicates additional code, which is not shown.

adiAssertSignal

Asserts an out-of-band signaling pattern to the line.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiAssertSignal** (CTAHD *ctahd*, unsigned *pattern*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>pattern</i>	Bit mask to assert.

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_AVAIL	Specified context's protocol does not allow an out-of-band signaling pattern.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function is not valid in the current port state.
CTAERR_OUTPUT_ACTIVE	Specified port is actively dialing.
CTAERR_SVR_COMM	Server communication error.

Details

adiAssertSignal asserts the specified out-of-band signaling *pattern*, which is either the physical out-of-band signal bits of digital lines or relates to the control of analog interface boards. In both cases, four signaling bits are addressed: A, B, C, and D, often written as ABCD, and defined by a bit mask (0x8, 0x4, 0x2, and 0x1, respectively). The following constants are in *adidef.h* and can be combined with the OR operator to assert any group of bits: ADI_A_BIT, ADI_B_BIT, ADI_C_BIT, and ADI_D_BIT.

This function cannot be used unless the current protocol specifically allows it. CTAERR_FUNCTION_NOT_AVAIL is returned if the application invokes **adiAssertSignal** when disallowed by the protocol.

When using **adiAssertSignal** and **adiStartPulse** on AG 4000, AG 4000C, AG 4040, AG 4040C, or CG boards configured for D4 framing, ensure that the C bit and the D bit are set the same as the A bit and the B bit. Otherwise, the received A bit and B bit at the remote end remains in an indeterminate state.

For example, to set the A bit high and the B bit low, call **adiAssertSignal** as follows:

```
adiAssertSignal (ctahd, ADI_A_BIT | ADI_C_BIT);
```


When using this function with a system that includes an analog interface board, refer to the hardware installation manual for the analog interface board for specific information on how the A and B bits affect the telephone line.

For more information, refer to *Performing low-level call control* on page 71.

See also

adiQuerySignalState, adiStartDial

Example

```
/*
 * Not using standard call control; managing actual line interface directly.
 */

#define MY_ONHOOK 0x00
#define MY_OFFHOOK ( ADI_A_BIT | ADI_B_BIT )

void myPickUp( CTAHD ctahd )
{
    adiAssertSignal( ctahd, MY_OFFHOOK );
}

void myHangUp( CTAHD ctahd )
{
    adiAssertSignal( ctahd, MY_ONHOOK );
}
```

adiCollectDigits

Starts collecting DTMF digits.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiCollectDigits** (CTAHD *ctahd*, char **buffer*, unsigned *maxdigits*, ADI_COLLECT_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>buffer</i>	Pointer to the buffer that receives the collected digits. Because the returned string is NULL-terminated, the <i>buffer</i> must be sized to at least <i>maxdigits</i> +1 bytes.
<i>maxdigits</i>	Maximum number of digits to collect.
<i>parms</i>	<p>Pointer to a digit collection parameter structure as shown (NULL designates default values for parameters):</p> <pre>typedef struct { DWORD size; /* size of this structure */ DWORD firsttimeout; /* timeout waiting for the first digit*/ /* use 0 to wait forever. */ DWORD intertimeout; /* timeout waiting for the next digit */ /* use 0 to wait forever. */ DWORD waitendtone; /* if non-zero, collection does not */ /* end until the end of the final dtmf*/ DWORD validDTMFs; /* mask of acceptable digits; use 0 */ /* or ADI_DIGIT_ANY to accept all. */ DWORD terminators; /* mask of terminating digits; use 0 */ /* to indicate no terminators. */ } ADI_COLLECT_PARMS;</pre> <p>Refer to <i>ADI_COLLECT_PARMS</i> on page 256 for field descriptions and valid values.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_BAD_ARGUMENT	buffer pointer is NULL.
CTAERR_BAD_SIZE	maxdigits is 0.
CTAERR_FUNCTION_ACTIVE	Attempt was made to get a digit or flush the digit queue while collecting digits.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function is not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_COLLECTION_DONE	<p>Generated when collection completes. The event buffer field points to the same buffer passed to adiCollectDigits. The size field contains the number of characters collected, plus one to account for the null terminator. The value field contains one of the following termination reasons, or an error code:</p> <p>CTA_REASON_DIGIT Terminating digit received.</p> <p>CTA_REASON_FINISHED Expected number of digits collected.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Terminated by adiStopCollection.</p> <p>CTA_REASON_TIMEOUT Timed out waiting for a digit.</p>

Details

Use **adiCollectDigits** to start the asynchronous collection of DTMF digits. Any digits received before collection is started are included, unless they were discarded by calling **adiFlushDigitQueue**. Any digit not included in the validDTMFs mask is discarded. Collection terminates and ADIEVN_COLLECTION_DONE is generated when one of the following occurs:

- The maximum number of digits (**maxdigits**) is collected.
- The initial (**firsttimeout**) or interdigit (**intertimeout**) timeout expires.
- A terminating (**terminators**) digit is received.
- **adiStopCollection** is issued.
- The call is released.

Note: If a digit is in both the terminators mask and in the validDTMFs mask, it is included as the last digit in the collected string. If the string contains **maxdigits** digits, the termination reason is CTA_REASON_FINISHED.

See also**adiGetDigit, adiStartDTMFDetector, adiStopDTMFDetector****Example**

```

int myGetDigits( CTAHD ctahd, char *digits, int maxdigits )
{
    ADI_COLLECT_PARMS parms;
    CTA_EVENT event;

    *digits = 0;

    ctaGetParms( ADI_COLLECT_PARMID, &parms, sizeof parms );
    parms.firsttimeout = 4000; /* wait 4 seconds for first digit */
    parms.intertimeout = 2000; /* wait 2 seconds between digits */

    adiCollectDigits( ctahd, digits, maxdigits, &parms );

while( 1 )
{
    myGetEvent( &event ); /* see ctaWaitEvent example */
    switch( event.id )
    {
        case ADIEVN_COLLECTION_DONE:
            if( event.value == CTA_REASON_RELEASED )
                return MYDISCONNECT; /* remote hang-up */
            else if( CTA_IS_ERROR( event.value ) )
                return MYFAILURE; /* AG Access API error */
            else if( strlen( digits ) == 0 )
                return MYFAILURE; /* no digits provided */
            else
                return SUCCESS; /* got digits */
            break;

        case ADIEVN_CALL_DISCONNECTED:
            /* In case this event was on the way up when we started
             * collection. Wait for 'collection done' event.
             */
            break;

        case ADIEVN_DIGIT_BEGIN:
        case ADIEVN_DIGIT_END:
            /* Typically don't want digit events. Wait for the
             * string of digits with 'collection done'.
             */
            break;
    }
}
}

```

adiCommandEchoCanceller

Sends a data array containing raw commands to an actively running echo cancellation function. Use **adiCommandEchoCanceller** to enable and configure echo canceller tone detection.

Supported board types

- AG
- CG

Prototype

DWORD **adiCommandEchoCanceller** (CTAHD *ctahd*, WORD **data* [], DWORD *nwords*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>data</i>	Pointer to an array of 16-bit data containing the commands.
<i>nwords</i>	Number of 16-bit data words.

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_FUNCTION_NOT_ACTIVE	Echo canceller function was not started before calling adiCommandEchoCanceller .
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_ECHOCANCEL_STATUS	Generated if the echo canceller enables send status mode. For information about this mode of operation, refer to echocancel.mode in ADI_START_PARMS. The echo canceller stores the status information in an event buffer. The information is arranged according to the ADI_ECHOCANCEL_STATUS_INFO structure in <i>adidef.h</i> . QX 2000 boards do not support the sending of echo canceller status information.
ADIEVN_ECHOCANCEL_TONE	This event contains two words of information about the tone that was detected. See Details for information.

DSP files

The following DSP file must be loaded to the board to enable echo canceller tone detection:

For these boards...	Add this DSP file...
AG	<i>echo_v4.m54</i>
CG	<i>echo_v4.f54</i>

Refer to the board installation and developer's manual for information about MIPS usage.

Details

The echo canceller must be started before **adiCommandEchoCanceller** will work. For information, see *Controlling echo* on page 57.

This function sends three commands to the tone detector.

Command A

Use the first command to start or stop the echo canceller tone detector, configure all options, and initialize all parameters:

Word	Description	Valid range	Typical values
Word 1	<p>Configures the echo canceller tone detector. Set bits as follows:</p> <ul style="list-style-type: none"> • Bit 0: 0 = stop, 1 = start • Bit 1: 0 = phase reversal requested • Bit 2: 0 = amplitude modulation detection requested • Bit 3: 0 = send/receive switching required • Bit 4: 0 = send path always if bit 3 = 1, 1 = receive path always if bit 3 = 1 • Bit 5: 0 = send all events to host • Bit 6: 0 = take direct control of echo canceller's NLP • Bit 7: 0 = take direct control of echo canceller's bypass • Bit 8: 0 = take direct control of echo canceller's reactivation after silence detection • Bit 9: 0 = silence detection locks the send/receive path when silence detected 	N/A	0001

Command B

Use the second command to specify the type of tone to be detected:

Word	Description	Valid range	Typical values
1	Tone number	0 through 3	N/A
2	Maximum frequency	[300 through 3000]	2100+15 for CED
3	Minimum frequency	[300 through 3000]	2100-15 for CED
4	Level (32). Compute the value using the following formula: $17030 \times \text{pow}(10, \text{level}/10.0)$, where level is in dBm in the range -42 to 0.	N/A	0001
5	Qualification time (ms)	0 through 32767	500 for CED

Tone number 0 is reserved for CED detection (phase reversal and amplitude modulation detection). Templates 1 through 3 can be set to any value. The template is deactivated when maximum frequency is set to 0.

Command C

Use the third command to modify the default configuration for CED detection, including phase reversal detection and amplitude modulation:

Word	Description	Valid range	Typical values
1	Maximum periodicity (ms) (phase reversal).	0 through 32767	480
2	Minimum periodicity (ms) (phase reversal).	0 through 32767	420
3	Maximum range of amplitude (amplitude modulation detection). A tolerance is taken by the program.	0 through 32767 (0 = no detection)	6554
4	Periodicity (amplitude modulation detection). The program on periodicity requirements takes a 20 to 25 percent tolerance.	0 through 32767	67
5	Time out before path switching (send/receive path switch). The DCE detector switches back and forth from send to receive until a tone is detected. This activity provides a 50 percent MIPS load for tone detection on both send and receive paths. The function needs about 50 ms to detect a tone and lock itself.	0 through 32767	60
6	Silence threshold (echo cancellation reactivation). Use the following formula to compute the value: $17030 \times \text{pow}(10, \text{level}/10.0)$, where level is in dBm in the range -42 to 0.	N/A	0021h
7	Silence duration. Since fax has a longer silence period, echo cancellation could be reactivated during fax protocol after the specified amount of time.	200 through 5000	NA

ADIEVN_ECHOCANCEL_TONE contains two words of information about the tone that was detected.

Word 1 indicates if the tone was a DCE tone or a single tone:

Bit number	Description
15	<p>Type of event.</p> <p>If bit 15 = 0 (DCE tone detected):</p> <ul style="list-style-type: none"> • Bit 14 = Amplitude modulation detected • Bit 13 = Phase reversals detected • Bit 12 = Silence detected after DCE detection <p>Examples:</p> <ul style="list-style-type: none"> • (14,13,12) = 0,0,0 (ANS) • (14,13,12) = 0,1,0 (ANS) • (14,13,12) = 1,0,0 (ANS_{am}) • (14,13,12) = 1,1,0 (ANS_{am}) • (14,13,12) = x,x,1 (silence after ANSxxx detection) <p>If bit 15 = 1 (single tone detected):</p> <ul style="list-style-type: none"> • Bits 14,13,12 = template's number detected <p>Examples:</p> <ul style="list-style-type: none"> • (14,13,12) = 0,0,1 (template 1) • (14,13,12) = 0,1,0 (template 2) • (14,13,12) = 0,1,1 (template 3)
11	<p>Path number on which tone was detected.</p> <ul style="list-style-type: none"> • 0 = Send path or echo canceller reference path • 1 = Receive path
4,3,2,1,0	<p>Level of the tone (dB). The tone level is equal to (bits 4-0) x (-3) dBm0. Dynamic goes from 0 to -93 dBm0. Precision is ± 1.5 dB.</p> <p>Examples:</p> <ul style="list-style-type: none"> • (0,0,0,1,1) = -9 dBm0 • (0,0,1,0,1) = -15 dBm0

Word 2 indicates the frequency (or frequencies) detected. Because templates can be programmed for a range of tones, it is possible to detect multiple tones within the same template. The following formula is used:

$$2 \times \cos(2\pi \times F/8000),$$

Examples:

- (15,...,0) = 7FFFh F = 0 Hz
- (15,...,0) = 3254h F = 1748 Hz
- (15,...,0) = 896Ah F = 2613 Hz

When a tone is detected, the program scans all templates and locks itself on the first template that satisfies the frequency and level detected. Program templates appropriately to deal with this behavior. For example:

- Template 2 = [800 - 1300] Hz, ...
- Template 3 = [1000 - 1020] Hz, ...

In this example, when a 1010 Hz tone appears, the program sends back an event associated with template 2 because it is the first template that meets all criteria for the detection. Thus, DCE detection is programmed on template 0 exclusively.

Once tone is detected, no path switching is performed until the end of detection.

If bit 9 of command A is set to 1, silence duration is computed according to the time connected to the right path. For example, if silence duration is set to 400 ms, 800 ms might pass before silence is detected.

See also**adiModifyEchoCanceller**

Example

```
// test routine that prompts for parameter values
DWORD myTestCommandEC (CTAHD ctahd)
{
    char command;
    WORD Aparams[1];
    WORD Bparams[5];
    WORD Cparams[7];

    DWORD nparms;
    WORD parms[16];
    WORD wtemp;
    DWORD dwtemp;

    promptchar ("Enter Command (A,B or C)", &command);
    switch (command)
    {
        case 'A':
            promptw ("EC Config", &Aparams[0]);
            nparms = 2;
            parms[0] = 1;    // command code "A"
            memcpy (&parms[1], Aparams, sizeof Aparams);
            break;
        case 'B':
            promptw ("Tone #", &Bparams[0]);
            promptw ("Max Freq", &Bparams[1]);
            promptw ("Min Freq", &Bparams[2]);
            prompt ("Level", &dwtemp);
            Bparams[3] = (WORD) (17030*pow(10,(dwtemp/10.0)));
            promptw ("Qual Time", &Bparams[4]);

            nparms = 6;
            parms[0] = 2;    // command code "B"
            memcpy (&parms[1], Bparams, sizeof Bparams);
            break;
        case 'C':
            promptw ("Phase rev: T_max (ms)", &Cparams[0]);
            promptw ("Phase rev: T_min (ms)", &Cparams[1]);
            promptw ("AM: factor (%)", &Cparams[2]);
            Cparams[2] = (WORD) (32767*wtemp/100);
            promptw ("AM: T (ms)", &Cparams[3]);
            promptw ("Tx/Rx path switching (ms)", &Cparams[4]);
            prompt ("Silence lvl (dB)", &dwtemp);
            Cparams[5] = (WORD) (17030*pow(10,(dwtemp/10.0)));
            promptw ("Silence duration (ms)", &Cparams[6]);

            nparms = 8;
            parms[0] = 3;    // command code "C"
            memcpy (&parms[1], Cparams, sizeof Cparams);
            break;
        default:
            printf("Invalid command\n");
            return -1;
    }
    return adiCommandEchoCanceller (ctahd, parms, nparms);
}
```

adiCommandRecord

Sends a data array containing raw commands to an actively running recording function. Use **adiCommandRecord** to enable and configure voice activity detection.

Supported board types

- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiCommandRecord** (CTAHD *ctahd*, WORD **data []*, DWORD *nwords*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>data</i>	Pointer to an array of 16-bit data containing the commands.
<i>nwords</i>	Number of 16-bit data words.

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	ADI service recording function was not started before calling adiCommandRecord .
CTAERR_INVALID_SEQUENCE	adiStopRecording has already been invoked.

Events

Event	Description
ADIEVN_RECORD_EVENT	<p>Contains information sent by the custom recording function. The event value field may contain one of the following reason codes (also defined in <i>/nms/include/evad.h</i>):</p> <p>EVAD_EVN_FUNCTION_DISABLED Voice activity detection disabled.</p> <p>EVAD_EVN_FUNCTION_ENABLED Voice activity detection enabled.</p> <p>EVAD_EVN_FUNCTION_ERROR Unknown or invalid parameter.</p> <p>EVAD_EVN_SIGNALLING_DISABLED Voice activity detection messaging disabled.</p> <p>EVAD_EVN_SIGNALLING_ENABLED Voice activity detection messaging enabled.</p> <p>EVAD_EVN_SPEECH_BEGIN Speech started. The event buffer contains the energy of the frame generating the event and the energy of the background noise in dB.</p> <p>EVAD_EVN_SPEECH_END Speech stopped. The event buffer contains the energy of the frame generating the event and the energy of the background noise in dB.</p> <p>EVAD_EVN_STREAMING_PAUSED Voice streaming from board to application paused.</p> <p>EVAD_EVN_STREAMING_RESUMED Voice streaming from board to application resumed.</p>

Note: The application receives ADIEVN_RECORD_EVENT asynchronously, while the speech buffers arrive every buffersize x framerate / framesize msec, attached to ADIEVN_RECORD_BUFFER_FULL (when speech is detected).

Details

The following DSP file must be loaded to the board to enable voice activity detection:

For these boards...	Add this DSP file...
AG	<i>rvoice_vad.m54</i>
CG	<i>rvoice_vad.f54</i>

To configure CG boards for voice activity detection, specify *rvoice_vad* in the resource definition. For example:

```
Resource[0].Definitions = (dtmf.det_all & rvoice_vad.rec_alaw & rvoice_vad.play_alaw...
```

To configure the PacketMedia HMP process for voice activity detection, set the value of the EnableVAD keyword to Yes. For example:

```
EnableVAD = Yes
```

To enable voice activity detection, call **adiCommandRecord** on an actively running ADI recording function (such as **adiRecordAsync**). Automatic gain control and energy detection must be disabled when using voice activity detection. Recording must be using ADI_ENCODE_MULAW, ADI_ENCODE_ALAW, or ADI_ENCODE_PCM8M16.

Call **adiCommandRecord** after receiving ADIEVN_RECORD_STARTED.

The first parameter must be one of the following voice activity detector commands:

Command	Description
EVAD_CDE_FUNCTION_ENABLE	Enable voice activity detection or update parameters. Default is disabled.
EVAD_CDE_FUNCTION_DISABLE	Disable voice activity detection. Silence is no longer suppressed.
EVAD_CDE_DEFAULT_ENABLE	Enable voice activity detection with default parameters.
EVAD_CDE_STREAMING_PAUSE	Pause sending voice data (silence or speech) to the host application. Useful for keeping voice activity detection energy thresholds update active when ASR is not active on the host.
EVAD_CDE_STREAMING_RESUME	Resume sending voice data to the host application.
EVAD_CDE_SIGNALLING_ENABLE	Send voice activity detection events to the host application (even if voice activity detection or record streaming are disabled). Default is disabled.
EVAD_CDE_SIGNALLING_DISABLE	Stop sending voice activity detection events (EVAD_SPEECH_BEGIN and EVAD_SPEECH_END) to the host application.

When enabling voice activity detection (EVAD_CDE_FUNCTION_ENABLE), modify the voice activity detector's default behavior with the following parameters (also defined in `/nms/include/evad.h`):

Parameter	Type	Default	Units	Description
snr	INT16	14	dB	Signal to noise ratio. Valid range is 5 to 30 dB.
hold_stop	INT16	1000	ms	Speech hangover time. Valid range is 300 to 2000 ms.
min_noise	INT16	-65	dB	Minimum noise floor. Valid range is -100 to -40 dB.
max_noise	INT16	-40	dB	Maximum noise floor. Valid range is -65 to -25 dB.
signal_attack	INT16	30	ms	Signal attack time constant. Valid range is 10 to 200 ms.
signal_release	INT16	60	ms	Signal release time constant. Valid range is 10 to 200 ms.
noise_attack	INT16	3000	ms	Noise attack time constant. Valid range is 500 to 5000 ms.
noise_release	INT16	600	ms	Noise release time constant. Valid range is 100 to 2000 ms.

Convert `signal_attack`, `signal_release`, `noise_attack`, and `noise_release` into DSP format using the following formula:

```
/* time constant (tc) and period need to have the same units of time */
int epsilon(float tc,float period)
{
    float eps;
    eps = (float)(1.0 - (float) exp((double)((-1.0 * period)/ tc)));
    return (int) (eps * 32767);    // return in S.15 format
}
LATENCY = 10; /* 10 msec record DPF period */
DSP_value = epsilon ( time_constant, LATENCY );
```

The custom recording DPF sends data to the host by calling the DSPOS function **dspkSendEvent**. The first parameter sent by the function displays in the value field of the CTA_EVENT structure. All remaining parameters display in an attached buffer. The application is responsible for freeing the buffer after it processes the data.

For more information, refer to *Detecting voice activity* on page 67.

Example

```

/* This code sends a command to custom record DPF and prints the events from the DPF.
*/
myWaitForEvent( ctaqueuehd, event );
switch(event->id)
{
    /* etc... */
    case ADIEVN_RECORD_STARTED:
    {
        WORD myParms[3] = { 0x1111, 0x2222, 0x3333 };
        adiCommandRecord( ctahd, myParms, 3);
        /* At the DSP level, the DPF will see the following
         * command packet.
         *
         * 0x3          -> size of command packet
         * 0x1111       -> parm 1
         * 0x2222       -> parm 2
         * 0x3333       -> parm 3
         */
    }
    case ADIEVN_RECORD_EVENT:
    {
        if (event->buffer != NULL) // event with multiple data
        {
            WORD i;
            WORD *pData = (WORD *) event->buffer;
            printf("event->value  %x\n", event->value );
            printf("event->size  %x\n", event->size );
            for (i=0; i < event->size / sizeof(WORD); i++)
            {
                printf("data[%d]  %x\n", i, pData[i]);
            }
            if (event->size & CTA_INTERNAL_BUFFER)
            {
                ctaFreeBuffer( event->buffer );
                printf("Buffer freed\n");
            }
        }
        else
        {
            // event with only 1 data
            printf("event->value  %x\n", event->value );
        }
        break;
    }
}

```

adiFlushDigitQueue

Flushes the internal digit collection queue.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiFlushDigitQueue** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_ACTIVE	Digit collection function is active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiFlushDigitQueue** to discard all digits in the ADI service internal digit collection queue. This function cannot be invoked while the application is actively collecting digits using **adiCollectDigits**.

If any digits are queued in the ADI service when a play or record voice operation is started, and the voice operation is to terminate on those specific touchtones, the voice operation terminates immediately. To prevent this from happening, use **adiFlushDigitQueue** or **adiGetDigit** to remove the escape key from the queue.

The digit queue is automatically flushed when a call is released.

For more information, refer to *Collecting digits* on page 53.

See also

adiPeekDigit, **adiStopCollection**

Example

```
/* Play a message, ignoring dtmfs. */
int myPlayToCompletion( CTAHD ctahd, unsigned encoding, void *buffer, unsigned bufsize )
{
    ADI_PLAY_PARMS playparms;
    CTA_EVENT event;

    ctaGetParms( ADI_COLLECT_PARMID, &playparms, sizeof playparms );
    playparms.DTMFabort = 0x0;

    adiPlayFromMemory( ctahd, encoding, buffer, bufsize, &playparms );

    do
    {
        myGetEvent( &event ); /* see ctaWaitEvent example */
    } while( event.id != ADIEVN_PLAY_DONE );

    if( event.value != CTA_REASON_FINISHED )
        return MYFAILURE;

    /* We've finished playing an uninterruptable message (no DTMF abort).
     * but some DTMFs may have been pressed and are sitting in the digit
     * collection queue. If we don't remove them, the queued digits
     * will cause the next interruptible play to be aborted immediately.
     */
    adiFlushDigitQueue( ctahd );
    return SUCCESS;
}
```


adiGetBoardInfo

Obtains information about a board.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetBoardInfo** (CTAHD *ctahd*, unsigned *board*, unsigned *size*, ADI_BOARD_INFO **boardinfo*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>board</i>	Board number as specified in the board keyword file.
<i>size</i>	Size of <i>boardinfo</i> structure.
<i>boardinfo</i>	Pointer to the ADI_BOARD_INFO structure, as shown: <pre>typedef struct { DWORD size; /* Size of this structure */ DWORD boardtype; /* Physical board type ADI_BOARDTYPE_XXX */ DWORD serial; /* Serial number */ DWORD ioaddr; /* Base IO address */ DWORD intnum; /* Interrupt number */ DWORD bufsize; /* Buffer size */ DWORD freemem; /* Available memory */ BYTE daughterboardid[4]; /* Daughterboard IDs 0 = none */ DWORD totalmips; /* Total gross DSP MIPS */ DWORD trunktype; /* Type of digital or analog trunk */ DWORD numtrunks; /* Number of trunks */ } ADI_BOARD_INFO;</pre>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>boardinfo</i> pointer is NULL.
CTAERR_BAD_SIZE	<i>size</i> is smaller than the size of DWORD.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetBoardInfo** to retrieve hardware configuration data for the specified board. The **board** argument identifies a particular board. This identifier must correlate to a board ID in the board keyword file. Refer to the board installation and developer's manual for more information.

Note: If an analog board is populated with a mixture of line interface types, the type of the lowest numbered interface is reported.

AG 4040 and AG 4040C boards are software compatible with AG 4000 and AG 4000C boards. When retrieving board information on AG 4040 or AG 4040C boards, **adiGetBoardInfo** reports the ADI board type as one of the AG 4000 or AG 4000C board types, for example, ADI_BOARDTYPE_AG4000_4T. The AG 4040 or AG 4040C trunk type, either T1 or E1, is configured in the board keyword file. If the trunk type is not specified, **adiGetBoardInfo** reports the ADI board type as one of the T1 variants.

The **ctahd** argument is used to access the context on which the ADI service was opened. The ADI service can be opened in driver-only mode if desired. In this case, no actual board resources are reserved. Set the board field in the MVIP_ADDR structure passed to **ctaOpenServices** to ADI_AG_DRIVER_ONLY. This function also works with a context that has the ADI service opened on actual MVIP streams and timeslots.

The **size** argument indicates how much memory to write at **boardinfo** address. The ADI service stores the actual number of bytes written in the ADI_BOARD_INFO size field.

See also

adiGetEEPromData

Example

```

void myShowBoardType( CTAHD ctahd, unsigned board )
{
    ADI_BOARD_INFO boardinfo;
    char            *type;
    unsigned        b_ports;
    int             ret;

    ret = adiGetBoardInfo( ctahd, board, sizeof boardinfo, &boardinfo );

    if( ret == SUCCESS )
    {
        switch( boardinfo.boardtype )

            case ADI_BOARDTYPE_QX2000 : type="QX 2000";b_ports=4;break;
            case ADI_BOARDTYPE_AG2000:  type="AG 2000"; b_ports= 8;break;
            case ADI_BOARDTYPE_AG4000_4T:type="AG 4000 4T"; b_ports=96;break;
            case ADI_BOARDTYPE_AG4000_4E:type="AG 4000 4E"; b_ports=120;break;
            case ADI_BOARDTYPE_CG6000C_QUAD:type="CG6000C_QUAD";b_ports=120;break;

            default:
            case ADI_BOARDTYPE_UNKNOWN : type="Unknown"; b_ports=0; break;
            }
        printf( "board:%2d at addr:%4x is an %-7s with %3d ports.\n",
                board, boardinfo.ioaddr, type, b_ports );
    }
    else if( ret == CTAERR_INVALID_BOARD )
        printf( "There is no board # %d.\n", board );
    else
        /* unexpected error */
        printf( "Error %x getting board # %d information.\n", ret, board );
}

```

adiGetBoardSlots

Returns the MVIP timeslots configured for the given board.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetBoardSlots** (CTAHD ***ctahd***, unsigned ***board***, unsigned ***mode***, unsigned ***maxslot***, ADI_TIMESLOT ****slotlist***, unsigned ****numslots***)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>board</i>	Board number as specified in the board keyword file.
<i>mode</i>	Stream capability, which can be either ADI_FULL_DUPLEX (both voice and signaling streams) or ADI_VOICE_DUPLEX (voice only).
<i>maxslot</i>	Maximum number of entries in <i>slotlist</i> array.
<i>slotlist</i>	Pointer to the ADI_TIMESLOT array, defined as: <pre>typedef struct { BYTE stream ; BYTE slot ; } ADI_TIMESLOT ;</pre>
<i>numslots</i>	Returned number of entries.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>slotlist</i> is NULL but <i>maxslot</i> is not 0 (zero), or <i>maxslot</i> is 0 (zero) but <i>slotlist</i> is not NULL, or <i>numslots</i> is NULL, or invalid <i>mode</i> .
CTAERR_INVALID_BOARD	<i>board</i> does not exist.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetBoardSlots** to query the available MVIP stream:slot pairs configured for a given board.

The **ctahd** argument is used to access the context on which the ADI service was opened. The ADI service can be opened in driver-only mode if desired. In this case, no actual board resources are reserved. Set the board field in the MVIP_ADDR structure passed to **ctaOpenServices** to ADI_AG_DRIVER_ONLY. This function also works with a context that has the ADI service opened on actual MVIP streams and timeslots.

If **mode** is 0 (zero), the slots returned are the DSP addresses that correspond to actual trunks, whether or not they are actually connected.

For example, if an AG 2000 board is partially populated, only the slots that contain line interfaces are returned.

Note: The DSPs are automatically connected to the trunk if the telephony bus is not enabled.

If **mode** is not 0 (zero), the function returns only those streams capable of supporting the given **mode**. The base stream for the given **mode** is returned in the ADI_TIMESLOT stream field.

Examples:

- For an AG 2000 board, stream 18 is voice and stream 19 is signaling. If **adiGetBoardSlots** is invoked with **mode** set to ADI_FULL_DUPLEX, the function returns an array of eight ADI_TIMESLOT structures, each with the stream field set to 18 (18:0..7).
- For an AG 2000 board, stream 18 is voice and stream 19 is signaling. If **adiGetBoardSlots** is invoked with **mode** set to ADI_VOICE_DUPLEX, the function returns an array of 16 ADI_TIMESLOT structures, each with the stream field set to 18 or 19 (18:0..7, 19:0..7).

For details on MVIP addressing, refer to the *Switching Service Developer's Reference Manual*.

The **maxslot** argument is the number of ADI_TIMESLOTS in the application supplied **slotlist** array. The ADI service returns the number of ADI_TIMESLOTS written to the **slotlist** in the **numslots** variable. This value is in the range 0 (zero) to **maxslot** inclusive.

Note: If **maxslot** is 0 (zero) and **slotlist** is NULL, **numslots** returns the actual number of slots without copying any data.

adiGetBoardSlots can be used with **adiGetBoardInfo** to dynamically configure an application's contexts. **ctaOpenServices** is called with a board number and MVIP stream:slot to open the ADI service. The application can retrieve a complete list of configured stream:slot pairs for any board with **adiGetBoardSlots**.

Example

```

#define MAX_SLOTS 256
void myShowBoardSlots( CTAHD ctahd, unsigned board )
{
    ADI_TIMESLOT slotlist [MAX_SLOTS]; /* Returned array of timeslots */
    int ret;
    unsigned stream, slot1, slot2, prevslot, numslots;

    /* Read the MVIP configuration for the board. */
    ret = adiGetBoardSlots( ctahd, board, ADI_VOICE_DUPLEX, MAX_SLOTS, slotlist, &numslots
);
    if( ret == SUCCESS )
    {
        /* The ADI_TIMESLOT information contains 'stream:slot' pairs.
        * Print the information as 'stream:slot0..slotN' ranges.
        */
        unsigned i = 0;
        while( i < numslots )
        {
            /* store stream and starting slot */
            stream = slotlist[i].stream;
            slot1 = slotlist[i].slot;
            prevslot = slot1;

            while( ++i < numslots && /* find ending slot */
                slotlist[i].stream == stream &&
                slotlist[i].slot == prevslot+1 )
                prevslot++;
            slot2 = slotlist[i-1].slot; /* store ending slot */

            printf( "%2d:%d", stream, slot1 );
            if( slot2 != slot1 ) printf("..%d", slot2 );
            puts( "" );
        }
    }
    else if( ret == CTAERR_INVALID_BOARD )
        printf( "There is no board # %d.\n", board );
    else
        /* unexpected error */
        printf( "Error %x getting board # %d information.\n", ret, board );
}

```

adiGetBoardSlots32

Returns the MVIP timeslots configured for the given board.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetBoardSlots32** (CTAHD *ctahd*, unsigned *board*, unsigned *mode*, unsigned *maxslot*, ADI_TIMESLOT32 **slotlist*, unsigned **numslots*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>board</i>	Board number as specified in the board keyword file.
<i>mode</i>	Stream capability, which can be either ADI_FULL_DUPLEX (both voice and signaling streams) or ADI_VOICE_DUPLEX (voice only).
<i>maxslot</i>	Maximum number of entries in <i>slotlist</i> array.
<i>slotlist</i>	Pointer to the ADI_TIMESLOT array, defined as: <pre>typedef struct { BYTE stream ; BYTE slot ; } ADI_TIMESLOT32 ;</pre>
<i>numslots</i>	Returned number of entries.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>slotlist</i> is NULL but <i>maxslot</i> is not 0 (zero), or <i>maxslot</i> is 0 (zero) but <i>slotlist</i> is not NULL, or <i>numslots</i> is NULL, or invalid <i>mode</i> .
CTAERR_INVALID_BOARD	<i>board</i> does not exist.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetBoardSlots32** to query the available MVIP stream:slot pairs configured for a given board.

The **ctahd** argument is used to access the context on which the ADI service was opened. The ADI service can be opened in driver-only mode if desired. In this case, no actual board resources are reserved. Set the board field in the MVIP_ADDR structure passed to **ctaOpenServices** to ADI_AG_DRIVER_ONLY. This function also works with a context that has the ADI service opened on actual MVIP streams and timeslots.

If **mode** is 0 (zero), the slots returned are the DSP addresses that correspond to actual trunks, whether or not they are actually connected.

For example, if an AG 2000 board is partially populated, only the slots that contain line interfaces are returned.

Note: The DSPs are automatically connected to the trunk if the telephony bus is not enabled.

If **mode** is not 0 (zero), the function returns only those streams capable of supporting the given **mode**. The base stream for the given **mode** is returned in the ADI_TIMESLOT32 stream field.

Examples:

- For an AG 2000 board, stream 18 is voice and stream 19 is signaling. If **adiGetBoardSlots32** is invoked with **mode** set to ADI_FULL_DUPLEX, the function returns an array of eight ADI_TIMESLOT32 structures, each with the stream field set to 18 (18:0..7).
- For an AG 2000 board, stream 18 is voice and stream 19 is signaling. If **adiGetBoardSlots32** is invoked with **mode** set to ADI_VOICE_DUPLEX, the function returns an array of 16 ADI_TIMESLOT32 structures, each with the stream field set to 18 or 19 (18:0..7, 19:0..7).

For details on MVIP addressing, refer to the *Switching Service Developer's Reference Manual*.

The **maxslot** argument is the number of ADI_TIMESLOT32 instances in the application supplied **slotlist** array. The ADI service returns the number of ADI_TIMESLOT32 instances written to the **slotlist** in the **numslots** variable. This value is in the range 0 (zero) to **maxslot** inclusive.

Note: If **maxslot** is 0 (zero) and **slotlist** is NULL, **numslots** returns the actual number of slots without copying any data.

adiGetBoardSlots32 can be used with **adiGetBoardInfo** to dynamically configure an application's contexts. **ctaOpenServices** is called with a board number and MVIP stream:slot to open the ADI service. The application can retrieve a complete list of configured stream:slot pairs for any board with **adiGetBoardSlots32**.

Example

```

#define MAX_SLOTS 480
void myShowBoardSlots( CTAHD ctahd, unsigned board )
{
    ADI_TIMESLOT32 slotlist [MAX_SLOTS]; /* Returned array of timeslots */
    int ret;
    unsigned stream, slot1, slot2, prevslot, numslots;

    /* Read the MVIP configuration for the board. */
    ret = adiGetBoardSlots32( ctahd, board, ADI_VOICE_DUPLEX, MAX_SLOTS, slotlist,
&numslots );
    if( ret == SUCCESS )
    {
        /* The ADI_TIMESLOT32 information contains 'stream:slot' pairs.
        * Print the information as 'stream:slot0..slotN' ranges.
        */
        unsigned i = 0;
        while( i < numslots )
        {
            /* store stream and starting slot */
            stream = slotlist[i].stream;
            slot1 = slotlist[i].slot;
            prevslot = slot1;

            while( ++i < numslots && /* find ending slot */
                slotlist[i].stream == stream &&
                slotlist[i].slot == prevslot+1 )
                prevslot++;
            slot2 = slotlist[i-1].slot; /* store ending slot */

            printf( "%2d:%d", stream, slot1 );
            if( slot2 != slot1 ) printf("..%d", slot2 );
            puts( "" );
        }
    }
    else if( ret == CTAERR_INVALID_BOARD )
        printf( "There is no board # %d.\n", board );
    else
        /* unexpected error */
        printf( "Error %x getting board # %d information.\n", ret, board );
}

```

adiGetContextInfo

Retrieves configuration information about a specified context.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetContextInfo** (CTAHD *ctahd*, ADI_CONTEXT_INFO **info*, unsigned *size*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>info</i>	<p>Pointer to a buffer to receive the information. The ADI_CONTEXT_INFO structure is shown:</p> <pre>typedef struct { DWORD size; /* User accessible CONTEXT INFO structure:*/ DWORD queueid; /* returned size of this structure */ DWORD userid; /* not used */ INT32 agliberr; /* last error code after calling AGLIB */ DWORD channel; /* AG Channel */ DWORD board; /* AG Board number */ DWORD stream; /* MVIP stream of this port */ DWORD timeslot; /* MVIP slot of this port */ DWORD mode; /* MVIP mode of operation of this port */ DWORD maxbufsize; /* maximum board buffer size */ char tcpname[12]; /* Current Protocol */ DWORD state; /* port state */ DWORD stream95; /* MVIP-95 base stream number */ } ADI_CONTEXT_INFO;</pre> <p>Refer to the Details section for a description of these fields.</p>
<i>size</i>	Amount of memory available at <i>info</i> , which must be large enough to receive the ADI_CONTEXT_INFO size return value.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>info</i> is NULL.
CTAERR_BAD_SIZE	<i>size</i> is smaller than the size of DWORD.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetContextInfo** to return information about the current state of a specified context.

Up to **size** bytes of the ADI_CONTEXT_INFO structure are copied to the address at **info**. If **size** is greater than or equal to sizeof (ADI_CONTEXT_INFO), the complete structure is copied. The number of bytes actually copied is returned in the ADI_CONTEXT_INFO size field.

Note: If you are using the Natural Call Control service, **adiGetContextInfo** does not fill in the tcpname field of the ADI_CONTEXT_INFO structure. To retrieve this information, the application must call **nccGetLineStatusInfo**.

The following table summarizes the ADI_CONTEXT_INFO structure. Many of these context characteristics are described in other functions, as noted:

Field	Description	Related functions
size	Returned size.	N/A
queueid	Not used.	N/A
userid	Not used.	N/A
agliberr	NMS internal.	N/A
channel	NMS internal.	N/A
board	Board number on which the context's DSP resides.	ctaCreateContext
stream	Base MVIP stream for the context.	ctaCreateContext
timeslot	Context's MVIP-90 timeslot.	ctaCreateContext
mode	Context's MVIP mode.	ctaCreateContext
maxbufsize	Board physical buffer size.	adiGetEncodingInfo
tcpname	Protocol executing on the context.	adiStartProtocol
state	Context state.	N/A
stream95	Base MVIP-95 stream.	N/A

Example

```
int myShowContextState( CTAHD ctahd )
{
    ADI_CONTEXT_INFO info;

    if( adiGetContextInfo( ctahd, &info, sizeof info ) != SUCCESS )
        return MYFAILURE;

    printf( " Queue ID = %d\n", info.queueid );
    printf( " User ID = %08Xh\n", info.userid );
    printf( " AG Channel = %08Xh\n", info.channel );
    printf( "Last AGLIB Error = %d \n", info.agliberr );
    printf( " AG Buffer Size = %d\n", info.maxbufsize );
    printf( " Protocol = %s\n", info.tcpname );
    printf( " Board Number = %d\n", info.board );
    printf( "Stream:Slot,Mode = %d:%d,", info.stream, info.timeslot );

    switch( info.mode )
    {
        case ADI_FULL_DUPLEX : puts("ADI_FULL_DUPLEX" ); break;
        case ADI_VOICE_DUPLEX : puts("ADI_VOICE_DUPLEX" ); break;
        case ADI_SIGNAL_DUPLEX : puts("ADI_SIGNAL_DUPLEX" ); break;
        default:
            if( info.mode & ADI_VOICE_INPUT ) printf( "+ADI_VOICE_INPUT" );
            if( info.mode & ADI_VOICE_OUTPUT ) printf( "+ADI_VOICE_OUTPUT" );
            if( info.mode & ADI_SIGNAL_INPUT ) printf( "+ADI_SIGNAL_INPUT" );
            if( info.mode & ADI_SIGNAL_OUTPUT ) printf( "+ADI_SIGNAL_OUTPUT" );
            printf( "\n" );
            break;
    }
    printf("\n");
    return SUCCESS;
}
```

adiGetDigit

Retrieves a digit from the front of the ADI service internal digit queue.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetDigit** (CTAHD *ctahd*, char **digit*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>digit</i>	Pointer to a character to store the digit copied from the digit queue. Valid digit values are the ASCII characters 0 through 9, # (number sign), and * (asterisk), as well as A, B, C, and D. If the digit queue is empty, <i>*digit</i> receives a value of 0 (zero).

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>digit</i> is a NULL pointer.
CTAERR_FUNCTION_ACTIVE	Digit collection function is already active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetDigit** to retrieve a single DTMF digit character from the front of the ADI service internal digit queue. The oldest digit is removed from the queue and copied to the address pointed to by *digit*. If the digit queue is empty, the value copied is 0 (zero).

The application must also be using **ctaWaitEvent** for digits to accumulate in the ADI service internal digit queue.

This function cannot be invoked if the application is actively collecting digits using **adiCollectDigits**.

To read the first digit without removing it from the collection queue, use **adiPeekDigit**.

If there is a digit in the internal digit queue that is configured in the *abort_mask* of a play or record operation to terminate the operation, the operation terminates immediately. Use **adiGetDigit** to remove the digit from the queue.

For more information, refer to *Collecting digits* on page 53.

See also**adiFlushDigitQueue, adiStopCollection****Example**

```
/* Remove and display digits in the digit queue */
void getandshowdigits( CTAHD ctahd )
{
    for (;;)
    {
        char digit;

        adiGetDigit( ctahd, &digit );
        if( digit == '\0' )
            break;
        putchar( digit );
    }
    putchar( '\n' );
}
```

adiGetEEPromData

Reads the on-board OEM data for a given board.

Supported board types

- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetEEPromData** (CTAHD *ctahd*, unsigned *board*, unsigned *size*, ADI_EEPROM_DATA **eeepromdata*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>board</i>	Board number as specified in the board keyword file.
<i>size</i>	Size of the caller's structure (the returned size is enclosed in the <i>eeepromdata</i> structure).
<i>eeepromdata</i>	Pointer to the returned structure, as shown: <pre>typedef struct { DWORD size; /* Size of this structure */ WORD data[32]; /* EEprom data */ } ADI_EEPROM_DATA;</pre>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	NULL pointer was passed for <i>eeepromdata</i> .
CTAERR_BAD_SIZE	<i>size</i> is smaller than the size of DWORD.
CTAERR_INVALID_BOARD	Invalid <i>board</i> was specified.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetEEPromData** to return OEM information stored on a PROM on the specified board.

The *ctahd* argument is used to access the context on which the ADI service was opened. The ADI service can be opened in driver-only mode if desired. In this case, no actual board resources are reserved. Set the board field in the MVIP_ADDR structure passed to **ctaOpenServices** to ADI_AG_DRIVER_ONLY. This function also works with a context that has the ADI service opened on actual MVIP streams and timeslots.

Note: QX 2000 boards do not support this function. If this function is called using a QX 2000 board, CTAERR_FUNCTION_NOT_AVAIL is returned.

See also**adiGetBoardInfo****Example**

```
/* Display first 16-bit value in EEPROM */
void showeeprom (unsigned drvid)
{
    ADI_EEPROM_DATA eeprom;

    adiGetEEPromData( drvid, 0, sizeof eeprom, &eeprom);
    printf("data[0] = %x\n", eeprom.data[0]);
}
```


adiGetEncodingInfo

Returns data size parameters for a given voice encoding format on a specified context.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetEncodingInfo** (CTAHD *ctahd*, unsigned *encoding*, unsigned **framesize*, unsigned **datarate*, unsigned **maxbufsize*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>encoding</i>	Data encoding method. See <i>Voice encoding formats</i> on page 13 for a complete list of valid encoding methods.
<i>framesize</i>	Pointer to returned size, in bytes, of a single voice frame for given encoding format.
<i>datarate</i>	Pointer to returned required throughput in bytes per second, for given encoding format.
<i>maxbufsize</i>	Pointer to returned board buffer size in bytes, for given encoding format on the specified context. For information specific to QX boards, refer to the <i>QX 2000 Installation and Developer's Manual</i> .

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid <i>encoding</i> format.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetEncodingInfo** to return data size information for the given encoding format on the specified context.

When submitting buffers of voice data for play or record, the buffers must be an integral multiple of the encoding frame size, and should be a multiple of the board's physical buffer (*maxbufsize*). All buffers of voice data submitted to the ADI service must be an integral number of *framesize* bytes. For example, if the frame size is 62 bytes, a submitted buffer must be sized as *n* x 62 bytes where *n*=1,2,3....

The ***datarate*** is provided for resource management optimization. The ***datarate*** defines the required throughput between the host CPU and AG board (in bytes/second). It can be used for positioning. For example, to skip ahead four seconds in a message, move your data pointer 4 x ***datarate*** bytes (modulo ***framesize***).

The ***maxbufsize*** is the maximum physical buffer size for the board on the specified context for the given encoding format. The board's physical buffer size varies depending upon the board type and configured software. The size returned here is rounded to a multiple of the frame size.

For information specific to QX boards, refer to the *QX 2000 Installation and Developer's Manual*.

You can pass NULL for any of the function arguments that are pointers to returned values.

See also

adiPlayAsync, adiPlayFromMemory, adiRecordAsync, adiRecordToMemory, adiStartPlaying, adiStartRecording

Example

```
void myShowEncodingInfo( CTAHD ctahd, unsigned encoding )
{
    unsigned framesize, datarate, maxbufsize;

    if( adiGetEncodingInfo( ctahd, encoding,
                            &framesize, &datarate, &maxbufsize ) == SUCCESS )
    {
        printf( "Frame size = %d bytes\n", framesize );
        printf( "Data rate = %d bytes/sec\n", datarate );
        printf( "Max buf size = %d bytes\n", maxbufsize );
    }
}
```

adiGetPlayStatus

Retrieves status for the active or most recently executed play operation.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetPlayStatus** (CTAHD *ctahd*, ADI_PLAY_STATUS **info*, unsigned *size*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>info</i>	<p>Pointer to the ADI_PLAY_STATUS structure, as shown:</p> <pre>typedef struct { DWORD size; /* returned size (GetPlayStatus()) */ DWORD reason; /* reason last play ended */ DWORD buffercount; /* counter of buffers submitted */ DWORD framecount; /* number of frames submitted */ DWORD totalbytes; /* total bytes submitted */ void *buffer; /* last buffer pointer submitted */ DWORD bytecount; /* size of last buffer submitted */ DWORD bytesplayed; /* total bytes actually played */ DWORD timestarted; /* actual time started (ms units) */ DWORD underrun; /* counts out-of-frame events */ } ADI_PLAY_STATUS;</pre> <p>Refer to the Details section for field descriptions.</p>
<i>size</i>	Amount of memory available at <i>info</i> to receive the ADI_PLAY_STATUS.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>info</i> is NULL.
CTAERR_BAD_SIZE	<i>size</i> is smaller than the size of DWORD.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetPlayStatus** to retrieve status information about the active or most recently completed voice play operation for the specified context.

Up to **size** bytes of the ADI_PLAY_STATUS structure are copied to the address pointed to by **info**. If **size** is greater than or equal to the size of ADI_PLAY_STATUS, the complete structure is copied. The number of bytes copied is returned in the ADI_PLAY_STATUS size field.

adiGetPlayStatus can be issued while actively playing. If there is no active play operation, the status information pertains to the most recently completed instance.

The ADI_PLAY_STATUS structure contains the following fields:

Field	Description
size	Number of bytes copied to info .
reason	Termination condition for the last ADIEVN_PLAY_DONE. This field is 0 if the play operation is active, or if it has not been started since the context was last opened.
buffercount	Number of buffers submitted.
framecount	Number of voice frames submitted.
totalbytes	Number of bytes submitted by the application.
buffer	Last buffer pointer submitted.
bytecount	Size of the last buffer submitted.
bytesplayed	Total number of bytes actually processed by the DSP and transmitted.
timestarted	Timestamp for when the play operation started. Refer to adiGetTimeStamp .
underrun	Total number of underruns during the play instance.

See also

adiPlayAsync, **adiPlayFromMemory**, **adiStartPlaying**, **adiStopPlaying**

Example

```
void myShowPlayStatus( CTAHD ctahd )
{
    ADI_PLAY_STATUS playstatus;

    adiGetPlayStatus( ctahd, &playstatus, sizeof playstatus );

    printf( "Termination condition=%x bytes played=%d\n",
           playstatus.reason, playstatus.bytesplayed );
}
```

adiGetRecordStatus

Retrieves the record operation status.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetRecordStatus** (CTAHD *ctahd*, ADI_RECORD_STATUS **info*, unsigned *size*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>info</i>	<p>Pointer to the ADI_RECORD_STATUS structure, as shown:</p> <pre>typedef struct { DWORD size ; /* Parms related to RECORD functions:*/ DWORD reason; /* Returned size (GetRecordStatus()) / DWORD buffercount; /* Reason last record ended */ DWORD frame; /* Counter of buffers submitted */ DWORD totalbytes; /* Number of frames submitted */ void *buffer; /* Total bytes submitted */ DWORD bytecount; /* Last buffer pointer SUBMITTED. */ DWORD bytesrecorded; /* Number of bytes into this buffer */ DWORD timestarted; /* Total bytes actually recorded. */ DWORD underrun; /* Actual time started (ms units) */ DWORD underrun; /* Counts underrun events */ } ADI_RECORD_STATUS;</pre> <p>Refer to the Details section for field descriptions.</p>
<i>size</i>	Amount of memory available at <i>info</i> to receive the ADI_RECORD_STATUS.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>info</i> is NULL.
CTAERR_BAD_SIZE	<i>size</i> is smaller than the size of DWORD.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetRecordStatus** to retrieve status information about the active or most recently completed voice record operation for the specified context.

Up to **size** bytes of the ADI_RECORD_STATUS structure are copied to the address pointed to by **info**. If **size** is greater than or equal to the size of ADI_RECORD_STATUS, the complete structure is copied. The number of bytes copied is returned in the ADI_RECORD_STATUS size field.

adiGetRecordStatus can be issued while actively recording. If there is no active record operation, the status information pertains to the most recently completed instance.

The ADI_RECORD_STATUS structure contains the following fields:

Field	Description
size	Number of bytes copied to info .
reason	Termination condition for the last ADIEVN_RECORD_DONE. This field is 0 if the record operation is active, or if it has not been started since the context was last opened.
buffercount	Number of buffers submitted.
framecount	Number of voice frames submitted.
totalbytes	Number of bytes submitted by the application.
buffer	Last buffer pointer submitted.
bytecount	Size of the last buffer submitted.
bytesrecorded	Total number of bytes received.
timestarted	Timestamp for the start of the record operation. Refer to adiGetTimeStamp .
underrun	Total number of underruns during the record instance.

See also

adiRecordAsync, **adiRecordToMemory**, **adiStartRecording**, **adiStopRecording**

Example

```
void myShowRecordStatus( CTAHD ctahd )
{
    ADI_RECORD_STATUS recordstatus;

    adiGetRecordStatus( ctahd, &recordstatus, sizeof recordstatus );

    /* A termination condition of 0 indicates either record in progress,
     * or none yet started in this CTA context.
     */
    printf( "Termination condition=%x, bytes recorded=%d\n", recordstatus.reason,
recordstatus.bytesrecorded );
}
```

adiGetTimeStamp

Converts an event timestamp to a count of the seconds elapsed since January 1, 1970.

Supported board types

- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiGetTimeStamp** (CTAHD *ctahd*, DWORD *msgtime*, unsigned long **timesec*, unsigned **timems*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>msgtime</i>	Event time stamp.
<i>timesec</i>	Pointer to returned seconds.
<i>timems</i>	Pointer to returned milliseconds.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiGetTimeStamp** to convert an event timestamp to a count of the number of seconds elapsed since 00:00:00 January 1, 1970. The *msgtime* is the CTA_EVENT timestamp value, which is in millisecond units with a 10-millisecond resolution. This function converts the *msgtime* into *timesec* seconds and *timems* milliseconds since midnight 1/1/70.

Because the event timestamp is 32 bits, it wraps every 2^{32} milliseconds (about 49 days). **adiGetTimeStamp** assumes the event occurred within 24 days.

Note: This function is not supported on a QX 2000 board. If this function is called using a QX 2000 board, CTAERR_FUNCTION_NOT_AVAIL is returned. Use **ctaGetTimeStamp** instead.

Example

```
#include <time.h>
void myShowTime( CTAHD ctahd, CTA_EVENT *event )
{
    struct tm *ptime;
    unsigned long timesec;
    unsigned timems;

    adiGetTimeStamp( ctahd, event->timestamp, &timesec, &timems );
    ptime = localtime( &timesec );
    printf( "%02d:%02d:%02d.%03d\n",
            ptime->tm_hour, ptime->tm_min, ptime->tm_sec, timems );
}
```


adiInsertDigit

Inserts a digit at the end of the ADI service internal digit queue.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiInsertDigit** (CTAHD *ctahd*, char *digit*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>digit</i>	Alphanumeric characters to store in the digit queue. Valid values are ASCII characters 0 through 9, # (number sign), and * (asterisk), as well as A, B, C, and D.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>digit</i> is not a valid character.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Protocol not started.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiInsertDigit** to insert a digit into the ADI digit queue. If digit collection is active, the digit is moved to the collection buffer and the interdigit timer is reset.

This function can be used when digits arrive from either DTMF detection or from an out-of-band indication such as RFC2833 packets. The DTMF digits are automatically added to the queue, whereas you must call this function to add the out-of-band digits.

The digit queue holds 62 characters. If the digit queue is full, the oldest character is discarded without an error indication.

If the digit is in the abort mask of an active play or record operation, the play or record operation terminates immediately. If any digit in the queue is in the abort mask, subsequent play or record operations terminate immediately after being started. Use **adiFlushDigitQueue**, **adiGetDigit**, or **adiCollectDigits** to remove digits from the queue.

For more information, refer to *Collecting digits* on page 53.

See also**adiPeekDigit****Example**

```
//This example shows Fusion RFC 2833 events being converted to ADI digits

#include "mspunsol.h"
#include "mspdef.h"

example(CTAHD ctahd)
{
//main event loop
for(;;)
{
CTA_EVENT event;
myGetEvent( &event );    /* see ctaWaitEvent example */
switch( event.id )
{
//Assumes RTP endpoint is configured with
// dtmf_event_control = SEND_FIRST_EVENT | SEND_LAST_EVENT

case MSPEVN_RFC2833_REPORT:
{
DISASM_DTMF_EVENT_STRUCT *dtmfEvt=
(DISASM_DTMF_EVENT_STRUCT *) (event.buffer);
if ((dtmfEvt->EvtVol & LAST_DTMF_EVENT)==0)
{
char digit='\0';
switch (DtmfEvt->EvtID)
{
case 0:
case 1:
case 2:
case 3:
case 4:
case 5:
case 6:
case 7:
case 8:
case 9: digit='0'+DtmfEvt->EvtID; break;
case 10: digit='*'; break;
case 11: digit='#'; break;
case 12: digit='A'; break;
case 13: digit='B'; break;
case 14: digit='C'; break;
case 15: digit='D'; break;
}
if (digit !='\0')
{
adiInsertDigit(ctahd, digit);
}
}
mspReleaseBuffer( event.objHd, event.buffer);
break;
}

// other events ...
}
}
}
```

adiModifyEchoCanceller

Modifies echo cancellation parameters after echo cancellation is started.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiModifyEchoCanceller** (CTAHD *ctahd*, ADI_ECHOCANCEL_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>parms</i>	<p>Pointer to echo cancellation parameters, as shown:</p> <pre>typedef struct { WORD size; /* parameters for echo cancellation */ DWORD mode; /* size of this structure */ DWORD filterlength; /* echo canceller mode */ DWORD adapttime; /* filter length (msec) */ DWORD predelay; /* filter adaptation time (msec) */ INT32 gain; /* offset of input sample (msec) */ } ADI_ECHOCANCEL_PARMS ;</pre> <p>For field descriptions and valid values, refer to <i>ADI_START_PARMS</i> on page 264.</p>

Return values

Return values	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_FUNCTION_NOT_ACTIVE	Echo canceller function was not started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_ECHOCANCEL_STATUS	Generated if the echo canceller enables send status mode. For more information about this mode of operation, refer to <i>echocancel.mode</i> in <i>ADI_START_PARMS</i> . The echo canceller stores the status information in an event buffer. The information is arranged according to the <i>ADI_ECHOCANCEL_STATUS_INFO</i> structure in <i>adidef.h</i> . QX 2000 boards do not support the sending of echo canceller status information.

Details

The following DSP file must be loaded to the board before running **adiModifyEchoCanceller**:

For these boards...	Load this DSP file...
AG	<i>echo.m54, echo_v3.m54, or echo_v4.m54</i>
CG	<i>echo.f54, echo_v3.f54, or echo_v4.f54</i>
QX 2000	The standard QX DSP file

Refer to *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for MIPS usage.

Use this function to modify echo cancellation parameters. The echo canceller must be started for **adiModifyEchoCanceller** to work. For more information, see *Controlling echo* on page 57.

Echo canceller operation can be enabled or disabled by setting the proper bits in the mode parameter. You can also change the gain applied to the near-end input and the predelay applied to the far-end input. You cannot change the filterlength and adapttime parameters.

You must always pass a pointer to the ADI_ECHOCANCEL_PARMS structure in the call to **adiModifyEchoCanceller** because the parameters for this function do not have default values. The echo cancel parameters are in the NCC.X.ADI_START.echocancel structure. You must copy the individual fields to the ADI_ECHOCANCEL_PARMS structure that you pass to **adiModifyEchoCanceller**.

For more information about the **adiModifyEchoCanceller** parameter fields, refer to *ADI_START_PARMS* on page 264.

ADI_ECHOCANCEL_STATUS_INFO structure

```
typedef struct
{
    WORD status;           /* Echo canceller status flags          */
    WORD ERL;              /* Echo Return Loss                     */
    WORD ERLE;             /* Echo Return Loss Enhancement         */
    WORD sndLevel;         /* Level of the sent signal             */
    WORD rcvLevel;         /* Level of the received signal         */
    WORD refPoint;         /* Reflection point location            */
} ADI_ECHOCANCEL_STATUS_INFO;
```

The ADI_ECHOCANCEL_STATUS_INFO structure contains the following fields:

Field	Description
status	Echo canceller status flags. See the status flag descriptions in the following table.
ERL	Echo return loss ratio. ERL is the ratio of rcvLevel to sndLevel. Compute the ERL in dBm as follows: $ERL_{dBm} = 10 \times \log (1/ERL)$
ERLE	Echo return loss enhancement. Compute the ERLE in dBm as follows: $ERLE_{dBm} = 10 \times \log (rcvLevel/(rcvLevel - ERLE))$
sndLevel	Power of the sent signal. Compute the sndLevel in dBm as follows: $sndLevel_{dBm} = 10 \times \log (sndLevel/0x3D29)$ where 0x3D29 is the 0 dBm reference value.
rcvLevel	Power of the received signal. Compute the rcvLevel in dBm as follows: $rcvLevel_{dBm} = 10 \times \log (rcvLevel/0x3D29)$ where 0x3D29 is the 0 dBm reference value.
refPoint	The position of the maximum value in the H register in 8 kHz sample increments. If the returned value of refPoint is 120, the reflection point is 15 ms, and a minimum tail length of 20 ms is required.

The following table describes the status flags:

Flag	Values
Status bits	
0	0 = Normal 1 = Send status one time
1	0 = Normal 1 = Send status automatically
2	0 = Enable HPF on reference stream (not used in v3 and up) 1 = Disable HPF on reference stream
3	0 = Disable comfort noise generation (used in v3 and up) 1 = Enable comfort noise generation 0 = Enable HPF on echo input stream (not used in v3 and up) 1 = Disable HPF on echo input stream
Control flags	
4	0 = Normal 1 = Reset filter taps to zero
5	0 = Normal 1 = Bypass echo canceler
6	0 = No adapt filter taps 1 = Adapt filter taps
7	0 = Enable NLP (echo suppressor) 1 = Disable NLP
Status flags	
8	0 = Diverged 1 = Converged
9	0 = Double talk 1 = Qualifying no double talk
10	0 = Double talk 1 = Qualified no double talk
11	0 = Not suppressing output 1 = Suppressing output
12	0 = Normal 1 = Possible double talk, but energy still within range of estimated ERL

See also

adiStartProtocol

Example

```
int myDisableEchoAdapt( CTAHD ctahd )
{
    ADI_ECHOCANCEL_PARMS echoParms = {0};
    NCC_ADI_START_PARMS nccStartParms = {0};

    /* get echo canceller parameters used by protocol for this ctahd */
    ctaGetParms ( ctahd, NCC_ADI_START_PARMID, &nccStartParms,
                  sizeof(nccStartParms) );

    echoParms.size =      sizeof(ADI_ECHOCANCEL_PARMS);
    echoParms.mode =      nccStartParms.echocancel.mode;
    echoParms.gain =       nccStartParms.echocancel.gain;
    echoParms.predelay =   nccStartParms.echocancel.predelay;

    echoParms.mode |= ADI_ECHOCANCEL_NO_ADAPT;

    if( adiModifyEchoCanceller( ctahd, &echoParms ) !=SUCCESS )
    {
        return MYFAILURE;
    }

    /* update the parameters */
    nccStartParms.echocancel.mode = echoParms.mode;
    ctaSetParamByName( ctahd, "ncc.x.adi_start", &nccStartParms,
                      sizeof nccStartParms);

    return MYSUCCESS;
}
```

adiModifyPlayGain

Sets the play gain for the duration of the active play operation.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiModifyPlayGain** (CTAHD *ctahd*, int *gain*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>gain</i>	The gain (dB) applied to the data as it is playing. The valid range is -54 through +24.

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Play operation is not currently active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiModifyPlayGain** to alter the gain applied to voice data as it is being transmitted. The *gain* remains set only for the current play operation instance. Values specified out of range are limited by the range.

See also

adiModifyPlaySpeed, **adiPlayAsync**, **adiPlayFromMemory**, **adiStartPlaying**, **adiStopPlaying**

Example

```

int myPlaySmartly( CTABD ctahd, unsigned encoding,
                  void *buffer, unsigned bufsize )
{
    ADI_PLAY_PARMS playparms;
    CTA_EVENT event;
    int currentgain = 0;
    int currentspeed = 100;

    ctaGetParms( ADI_COLLECT_PARMID, &playparms, sizeof playparms );
    playparms.DTMFabort = 0x0; /* Don't abort on any DTMF */

    if( adiPlayFromMemory( ctahd, encoding,
                          buffer, bufsize, &playparms ) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event ); /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_DIGIT_BEGIN:
                switch( (char)event.value )
                {
                    case '1': /* slower */
                        currentspeed -= 20;
                        adiModifyPlaySpeed( ctahd, currentspeed );
                        break;

                    case '3': /* faster */
                        currentspeed += 20;
                        adiModifyPlaySpeed( ctahd, currentspeed );
                        break;

                    case '4': /* softer */
                        currentgain -= 3; /* decrement 3 dB */
                        adiModifyPlayGain( ctahd, currentgain );
                        break;

                    case '6': /* louder */
                        currentgain += 3; /* increment 3 dB */
                        adiModifyPlayGain( ctahd, currentgain );
                        break;

                    default: /* ignore others */
                        break;
                }
                break;

            case ADIEVN_DIGIT_END: /* ignore end of dtmf */
            case 0: /* no event */
            default:
                break;
        }
    } while( event.id != ADIEVN_PLAY_DONE );

    if( event.value != CTA_REASON_FINISHED )
        return MYFAILURE;

    return SUCCESS;
}

```

adiModifyPlaySpeed

Sets the playback speed for the duration of the active play operation.

Supported board types

- AG
- CG

Prototype

DWORD **adiModifyPlaySpeed** (CTAHD *ctahd*, int *speed*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>speed</i>	Percentage of change to apply to the original recording, where 100 percent is no change. Valid range of change depends on the capabilities of the hardware and DSP files installed.

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Play is not currently active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiModifyPlaySpeed** to alter the speed (faster or slower) applied to voice data as it is being transmitted. The *speed* remains set only for the current play operation instance. Values specified out of range are limited by the range (the valid AG and CG board range is 100 to 200 percent).

Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Note: The PacketMedia HMP process and QX 2000 boards do not support this function. If this function is called using a QX 2000 board, CTAERR_FUNCTION_NOT_AVAIL is returned. If this function is called using PacketMedia HMP, it has no effect on the speed and does not return an error message.

See also

adiModifyPlayGain, **adiPlayAsync**, **adiPlayFromMemory**, **adiStartPlaying**, **adiStopPlaying**

adiPeekDigit

Reads the first digit in the ADI service internal digit queue without removing it.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiPeekDigit** (CTAHD *ctahd*, char **digit*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>digit</i>	Pointer to a character to store the digit copied from the digit queue.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>digit</i> is a NULL pointer.
CTAERR_FUNCTION_ACTIVE	Digit collection function is already active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

adiPeekDigit reads a single DTMF digit character from the front of the ADI service internal digit queue without removing it. The digit is copied to the address pointed to by *digit*. Valid digit values are the ASCII characters 0 through 9, # (number sign), and * (asterisk), as well as A, B, C, and D. If the digit queue is empty, the value is 0 (zero).

This function cannot be invoked if the application is actively collecting digits using **adiCollectDigits**.

For more information, refer to *Collecting digits* on page 53.

See also

adiFlushDigitQueue, **adiStopCollection**

adiPlayAsync

Initiates a voice play operation with asynchronous buffer submission.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiPlayAsync** (CTAHD *ctahd*, unsigned *encoding*, void **buffer*, unsigned *bufsize*, unsigned *flags*, ADI_PLAY_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>encoding</i>	Data encoding format. See <i>Voice encoding formats</i> on page 13 for a complete list.
<i>buffer</i>	Pointer to initial voice data buffer.
<i>bufsize</i>	Number of bytes stored at the address in <i>buffer</i> .
<i>flags</i>	Indicates if the specified buffer is the only buffer to be played (can be set to ADI_PLAY_LAST_BUFFER or 0).
<i>parms</i>	<p>Pointer to play parameters according to the following structure (NULL value uses the default play parameters):</p> <pre>typedef struct { DWORD size; /* size of this structure */ DWORD DTMFabort; /* abort on DTMF */ INT32 gain; /* playing gain in dB */ DWORD speed; /* initial speed in percent (AG boards only) */ DWORD maxspeed; /* max play speed in percent (AG boards only) */ } ADI_PLAY_PARMS;</pre> <p>Refer to <i>ADI_PLAY_PARMS</i> on page 261 for field descriptions and valid values.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_BAD_ARGUMENT	Either invalid <i>encoding</i> or NULL <i>buffer</i> .
CTAERR_BAD_SIZE	<i>bufsize</i> is not a multiple of framesize for selected <i>encoding</i> .
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_OUTPUT_ACTIVE	Play failed because there is another active output function.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_PLAY_BUFFER_REQ	Generated when the ADI service needs a buffer containing voice data. The application responds by either submitting a full buffer (adiSubmitPlayBuffer) or a full or partial buffer (adiSubmitPlayBuffer with flag indicating ADI_PLAY_LAST_BUFFER). If the ADI_PLAY_UNDERRUN bit is set, an underrun occurred, meaning that playing was temporarily suspended because there was no buffer to play.
ADIEVN_PLAY_DONE	Generated by the ADI service when the play operation terminates. The event size field contains the total number of bytes played during the function's instance. The event value field contains one of the following termination conditions, or an error code: CTA_REASON_DIGIT Aborted due to DTMF. CTA_REASON_FINISHED Buffer submitted with the ADI_PLAY_LAST_BUFFER flag set completed playing. CTA_REASON_RECOGNITION Aborted because of speech recognition. You receive this reason only if the application is using a speech recognition library. CTA_REASON_RELEASED Call terminated. CTA_REASON_STOPPED Stopped by application request.

Details

Use **adiPlayAsync** to initiate a voice playback operation. The voice data is supplied in a sequence of buffers. The application has complete latitude and responsibility for allocating, filling, and submitting buffers to the ADI service.

The **bufsize** can be arbitrarily large but must be an integral multiple of framesize bytes for the selected **encoding**. For optimum performance, the **bufsize** must be the largest frame multiple that fits in one board buffer. You can obtain this size by calling **adiGetEncodingInfo** (refer to the **maxbufsize** argument). If **bufsize** is less than or equal to the board buffer size, you can reuse the buffer as soon as this function returns. Otherwise, to avoid overwriting data, you must wait for the second ADIEVN_PLAY_BUFFER_REQ before you can reuse the buffer.

After play initiates, the ADI service sends ADIEVN_PLAY_BUFFER_REQ to the application whenever more data is needed. The application responds to this event by submitting a filled voice buffer with **adiSubmitPlayBuffer**. The application must submit buffers only in response to ADIEVN_PLAY_BUFFER_REQ.

For proper operation, each buffer must be submitted while the previous buffer is being played. If a buffer is submitted too late, an underrun occurs and silence is played. You can monitor for underruns by checking the ADI_PLAY_UNDERRUN bit in the value field of the ADIEVN_PLAY_BUFFER_REQ event. Use **adiGetPlayStatus** to retrieve a count of underruns that occurred since play started.

The application terminates play by submitting a buffer with the **flags** argument set to ADI_PLAY_LAST_BUFFER. After the ADI service has played the buffer that was submitted with the flag set, it generates ADIEVN_PLAY_DONE with the value set to CTA_REASON_FINISHED.

Refer to *Recording and playing* on page 13 for information about play operations in asynchronous mode.

Encoding formats and DSP files

When recording or playing speech files on AG boards, a specific DSP file must be loaded for each encoding type. For QX boards, the standard DSP file supports the valid encoding types. For more information, refer to *Voice encoding formats* on page 13.

When recording or playing speech files on CG boards, a specific DSP file must be loaded for each encoding type except when using the native play and record feature. The native play and record feature combines an ADI port with an MSPP endpoint and plays or records speech data directly to or from an IP endpoint with no transcoding. For information on the native play and record feature, refer to *Performing NMS native play and record* on page 31.

The following table lists the DSP files that must be loaded on the AG and CG boards. It also lists the valid encoding types that QX boards and PacketMedia HMP processes support:

Encoding type	AG DSP file	CG DSP file	QX support	PacketMedia HMP support
ADI_ENCODE_ALAW	<i>rvoice.m54</i> or <i>rvoice_vad.m54</i>	<i>rvoice.f54</i> or <i>rvoice_vad.f54</i>	Y	Y
ADI_ENCODE_G723_5		<i>g723.f54</i>	N	N
ADI_ENCODE_G723_6		<i>g723.f54</i>	N	N
ADI_ENCODE_G726	<i>g726.m54</i>	<i>g726.f54</i>	Y	Y
ADI_ENCODE_G726_16			y	N
ADI_ENCODE_G726_24			y	N
ADI_ENCODE_G726_32			y	N
ADI_ENCODE_G726_40			y	N
ADI_ENCODE_G729A		<i>g729.f54</i>	N	N
ADI_ENCODE_GSM	<i>gsm.ms.m54</i>	<i>gsm.ms.f54</i>	N	N
ADI_ENCODE_IMA_24	<i>ima.m54</i>	<i>ima.f54</i>	Y	N
ADI_ENCODE_IMA_32	<i>ima.m54</i>	<i>ima.f54</i>	Y	Y
ADI_ENCODE_NMS_16	<i>voice.m54</i>	<i>voice.f54</i>	Y	Y
ADI_ENCODE_NMS_24	<i>voice.m54</i>	<i>voice.f54</i>	Y	Y
ADI_ENCODE_NMS_32	<i>voice.m54</i>	<i>voice.f54</i>	Y	Y
ADI_ENCODE_NMS_64	<i>voice.m54</i>	<i>voice.f54</i>	Y	Y
ADI_ENCODE_MULAW	<i>rvoice.m54</i> or <i>rvoice_vad.m54</i>	<i>rvoice.f54</i> or <i>rvoice_vad.f54</i>	Y	Y
ADI_ENCODE_OKI_24	<i>oki.m54</i>	<i>oki.f54</i>	Y	N
ADI_ENCODE_OKI_32	<i>oki.m54</i>	<i>oki.f54</i>	Y	Y

Encoding type	AG DSP file	CG DSP file	QX support	PacketMedia HMP support
ADI_ENCODE_PCM8M16	<i>rvoice.m54</i> or <i>rvoice_vad.m54</i>	<i>rvoice.f54</i> or <i>rvoice_vad.f54</i>	Y	Y
ADI_ENCODE_PCM11M8	<i>wave.m54</i>	<i>wave.f54</i>	Y	N
ADI_ENCODE_PCM11M16	<i>wave.m54</i>	<i>wave.f54</i>	Y	N
ADI_ENCODE_VOX_32			y	N

Refer to *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for MIPS usage.

See also

adiModifyPlayGain, adiModifyPlaySpeed, adiPlayFromMemory, adiSetNativeInfo, adiStartPlaying, adiStopPlaying,

Example

Refer to the *playrec* demonstration program.

adiPlayFromMemory

Initiates a voice play operation using data from a single memory-resident buffer.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiPlayFromMemory** (CTAHD *ctahd*, unsigned *encoding*, void **buffer*, unsigned *bufsize*, ADI_PLAY_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>encoding</i>	Encoding type. See <i>Voice encoding formats</i> on page 13 for a complete list.
<i>buffer</i>	Pointer to voice data buffer.
<i>bufsize</i>	Number of bytes stored at the address in <i>buffer</i> (<i>bufsize</i> can be arbitrarily large).
<i>parms</i>	<p>Pointer to play parameters according to the following structure (NULL uses default values):</p> <pre>typedef struct { /* parms related to adiStartPlaying*/ DWORD size ; /* size of this structure */ DWORD DTMFabort; /* abort on DTMF; */ INT32 gain; /* Recording gain in dB */ DWORD speed; /* initial speed in percent */ DWORD maxspeed; /* maximum play speed in percent */ } ADI_PLAY_PARMS;</pre> <p>Note: Fields in bold are not applicable to the native play and record feature. Refer to <i>ADI_PLAY_PARMS</i> on page 261 for field descriptions and valid values.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_BAD_ARGUMENT	Either invalid <i>encoding</i> or NULL <i>buffer</i> .
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_OUTPUT_ACTIVE	Play failed because there is another active output function.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_PLAY_DONE	<p>Generated by the ADI service when playing terminates. The event size field contains the total number of bytes played during the function instance. The event value field contains one of the following terminating conditions, or an error code:</p> <p>CTA_REASON_DIGIT Aborted due to DTMF.</p> <p>CTA_REASON_FINISHED Complete buffer played.</p> <p>CTA_REASON_RECOGNITION Aborted because of speech recognition.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p>

Details

When recording or playing speech files on AG boards, a specific DSP file must be loaded for each encoding type. For QX boards, the standard DSP file supports the valid encoding types. For more information, refer to *Voice encoding formats* on page 13.

When recording or playing speech files on CG boards, a specific DSP file must be loaded for each encoding type except when using the native play and record feature. The native play and record feature combines an ADI port with an MSPP endpoint and plays or records speech data directly to or from an IP endpoint with no transcoding. For information on the native play and record feature, refer to *Performing NMS native play and record* on page 31.

For more information, see *Encoding formats and DSP files* on page 134. The table lists the DSP files that must be loaded on the AG and CG boards. It also lists the valid encoding types that QX boards and PacketMedia HMP processes support. Refer to the board installation and developer's manual for MIPS usage.

adiPlayFromMemory starts playing a single memory-resident buffer of **bufsize** bytes. The ADI service generates ADIEVN_PLAY_DONE when the function terminates. To avoid unintentionally modifying data, the application must not modify the buffer until it receives the DONE event.

For more information, refer to *Playing* on page 25.

See also

adiGetEncodingInfo, adiGetPlayStatus, adiModifyPlayGain, adiModifyPlaySpeed, adiPlayAsync, adiSetNativeInfo, adiStartPlaying, adiStopPlaying

Example

```
int myPlayMemory( CTAHD ctahd, unsigned encoding,
                 void *buffer, unsigned bufsize )
{
    CTA_EVENT event;

    if( adiPlayFromMemory( ctahd, encoding, buffer, bufsize, NULL ) != SUCCESS)
        return MYFAILURE;

    do
    {
        myGetEvent( &event ); /* see ctaWaitEvent example*/
    } while( event.id != ADIEVN_PLAY_DONE );

    if( event.value == CTA_REASON_RELEASED )
        return MYDISCONNECT; /* call has been terminated*/
    else if( CTA_IS_ERROR( event.value ) )
        return MYFAILURE;    /* API error */
    else
        return SUCCESS;      /* stopped normally */
}
```

adiQuerySignalState

Queries the current state of the out-of-band signaling bits. Use only with NOCC protocol.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiQuerySignalState** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	adiStartSignalDetector was not called.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_RESOURCE_CONFLICT	A protocol other than NOCC is active.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_QUERY_SIGNAL_DONE	After the ADI service queries the board for the current signaling pattern, the ADI service generates an event with the size field containing the current signaling pattern. It is a mask of the following constants found in <i>adidef.h</i> : ADI_A_BIT, ADI_B_BIT, ADI_C_BIT, and ADI_D_BIT.

Details

The AG 2000, AG 2000C, and AG 2000-BRI boards require *signal.m54* to be loaded.

Use **adiQuerySignalState** to query the out-of-band signaling detector for the current state of the signaling bits. These signaling bits can be the actual T1/E1 digital carrier signaling bits, or they can relate to specific detectors of analog interface boards (for example, a ring detector). In both cases, the ADI service recognizes four signaling bits: A, B, C, and D, often written as ABCD, and defined by the constants ADI_A_BIT, ADI_B_BIT, ADI_C_BIT, and ADI_D_BIT.

Note: This function can be called only if you started detection using **adiStartSignalDetector**.

Example

```
int myShowMVIP( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiQuerySignalState( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event ); /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_QUERY_SIGNAL_DONE:
                printf( "MVIP signalling bits = 0x%x (%c%c%c%c)\n",
                    (event.value&0xf),
                    (event.value&0x8)?'A':'-', (event.value&0x4)?'B':'-',
                    (event.value&0x2)?'C':'-', (event.value&0x1)?'D':'-' );
                break;

            /* ... */
        }
    }
}
```

adiRecordAsync

Initiates recording in asynchronous buffer mode.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiRecordAsync** (CTAHD **ctahd**, unsigned **encoding**, unsigned **maxmsec**, void ***buffer**, unsigned **bufsize**, ADI_RECORD_PARMS ***parms**)

Argument	Description
ctahd	Context handle returned by ctaCreateContext or ctaAttachContext .
encoding	Encoding type. See <i>Voice encoding formats</i> on page 13 for a complete list.
maxmsec	Maximum duration for recording (milliseconds). When voice activity detection is enabled, maxmsec is the maximum duration of speech recording, excluding silences.
buffer	Pointer to buffer to receive recorded data.
bufsize	Number of bytes available at buffer (bufsize must be set to an exact multiple of the framesize for the selected encoding).
parms	<p>Pointer to record parameters according to the following structure (NULL uses default values):</p> <pre>typedef struct { WORD size; /* size of this structure */ DWORD DTMFabort; /* mask that specifies DTMF tones to abort; */ INT32 gain; /* recording gain in dB SLC parms (used if silence det); */ DWORD novoicetime; /* length of initial silence to stop recording (ms); */ /* use 0 to deactivate initial silence detection. */ DWORD silencetime; /* length of silence to stop recording after */ /* voice has been detected; use 0 to deactivate. */ INT32 silenceampl; /* qualif level for silence (dBm) */ DWORD silencedeglitch; /* deglitch while qualifying silence */ /*--[Beep for record]----- */ DWORD beepfreq; /* beep frequency (Hz) */ INT32 beepampl; /* beep amplitude (dBm) */ DWORD beepetime; /* beep time (ms) 0=no beep */ /*--[AGC parms]----- */ DWORD AGCenable; /* enable AGC; use 1 to activate */ INT32 AGCtargetampl; /* target AGC level (dBm) */ INT32 AGCsilenceampl; /* silence level (dBm) */ WORD AGCattacktime; /* attack time (ms) */ DWORD AGCdecaytime; /* decay time (ms) */ } ADI_RECORD_PARMS;</pre> <p>Refer to ADI_RECORD_PARMS on page 262 for field descriptions and valid values.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_BAD_ARGUMENT	Either invalid encoding selected or NULL buffer pointer passed.
CTAERR_BAD_SIZE	size is less than one frame.
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_OUTPUT_ACTIVE	Record failed because there is another active output function.
CTAERR_RESOURCE_CONFLICT	Silence detector is in use by adiStartEnergyDetector .
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_RECORD_STARTED	Generated by the ADI service after the function is started on the board. If the ADI_RECORD_BUFFER_REQ bit in the event value field is set, more buffers are needed and the application must submit another empty buffer.
ADIEVN_RECORD_BUFFER_FULL	Generated by the ADI service when a buffer is filled with recorded voice data. The event contains the following fields: <ul style="list-style-type: none"> • buffer: Pointer to a previously submitted user buffer. • size: Number of bytes recorded into buffer. • value: Flags; If the ADI_RECORD_BUFFER_REQ bit is set, more buffers are needed and the application must submit another empty buffer. If the ADI_RECORD_UNDERRUN bit is set, an underrun occurred. There was no new buffer to record information when this one was completed.
ADIEVN_RECORD_DONE	Generated when the record operation completes. The event size field contains the total number of bytes recorded during the record instance lifetime. The value field contains one of the following termination reasons or error codes: <p>CTA_REASON_DIGIT Aborted due to DTMF.</p> <p>CTA_REASON_NO_VOICE No voice detected.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p> <p>CTA_REASON_TIMEOUT Record time limit (maxmsec) reached.</p> <p>CTA_REASON_VOICE_END User stopped speaking.</p> <p>CTAERR_FUNCTION_NOT_AVAIL Required DSP file not loaded on the board.</p> <p>CTAERR_xxx or ADIERR_xxx Record failed.</p>

Details

When recording or playing speech files on AG boards, a specific DSP file must be loaded for each encoding type. For QX boards, the standard DSP file supports the valid encoding types. For more information, refer to *Voice encoding formats* on page 13.

When recording or playing speech files on CG boards, a specific DSP file must be loaded for each encoding type except when using the native play and record feature. The native play and record feature combines an ADI port with an MSPP endpoint and plays or records speech data directly to or from an IP endpoint with no transcoding. For information on the native play and record feature, refer to *Performing NMS native play and record* on page 31.

For more information, see *Encoding formats and DSP files* on page 134. The table lists the DSP files that must be loaded on the AG and CG boards. It also lists the valid encoding types that QX boards and PacketMedia HMP processes support. Refer to the board installation and developer's manual for MIPS usage.

Use **adiRecordAsync** to initiate a voice record operation. The data is supplied to the application in a sequence of buffers. The application submits empty buffers using **adiSubmitRecordBuffer** for the duration of the operation. These buffers are then filled with recorded voice data and ADIEVN_RECORD_BUFFER_FULL events are returned. The application has complete latitude and responsibility for allocating, flushing, and submitting the buffers.

When the ADI service needs another buffer, it sets the ADI_RECORD_BUFFER_REQ bit in the event value field for ADIEVN_RECORD_STARTED and ADIEVN_RECORD_BUFFER_FULL. The application responds by submitting another empty buffer using **adiSubmitRecordBuffer**. The application submits buffers only when requested by the ADI service. The ADI service owns the buffer until either ADIEVN_RECORD_BUFFER_FULL or ADIEVN_RECORD_DONE is delivered to the application.

The last buffer before the DONE event can be a partial buffer. The DONE event itself does not include a buffer of data. The record operation terminates when the application receives ADIEVN_RECORD_DONE.

Note: The final buffer submitted is not always returned to the application. If the application dynamically allocates buffers, it must keep track of submitted buffers to free any outstanding buffers when record is done.

For optimum performance, the **bufsize** must be the largest frame multiple that fits in one board buffer. You can obtain this size by calling **adiGetEncodingInfo** (refer to the **maxbufsize** argument).

For proper operation, each buffer must be submitted while the previous buffer is being filled. If a buffer is submitted too late, an underrun occurs and the input data is lost. You can monitor for underruns by checking the ADI_RECORD_UNDERRUN bit in the value field of ADIEVN_RECORD_BUFFER_FULL. Use **adiGetRecordStatus** to retrieve a count of underruns that occurred since record started.

Note: You cannot initiate a record operation while playing voice or generating tones unless you disable the record beep by setting either ADI_RECORD.beeptime or ADI_RECORD.beepfreq to 0 (zero). You cannot start a record operation if the energy detector is active unless both ADI_RECORD.novoicetime and ADI_RECORD.silencetime are 0 (zero).

For more information, refer to *Recording* on page 20.

See also

adiCommandRecord, adiRecordToMemory, adiSetNativeInfo, adiStartRecording, adiStopRecording

Example

Refer to the *playrec* demonstration program.

adiRecordToMemory

Initiates recording of an RTP stream into a single memory-resident buffer.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiRecordToMemory** (CTAHD *ctahd*, unsigned *encoding*, void **buffer*, unsigned *bufsize*, ADI_RECORD_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>encoding</i>	Encoding type. See <i>Voice encoding formats</i> on page 13 for a complete list.
<i>buffer</i>	Pointer into process memory to receive encoded data.
<i>bufsize</i>	<p>Number of bytes pointed by <i>buffer</i> (<i>bufsize</i> can be arbitrarily large and is truncated to a multiple of the framesize for the selected <i>encoding</i>).</p> <p>If recording a channel using the native record feature and silence compression is enabled (refer to the <i>expandsilence</i> parameter in ADI_NATIVE_PARMS), this buffer size does not imply a specific time limit. If the application requires a specific time limit, use adiStartRecording or adiRecordAsync to set the maximum record time parameter.</p>
<i>parms</i>	<p>Pointer to record parameters according to the following structure (NULL designates default values):</p> <pre>typedef struct { DWORD size; /* Size of this structure */ DWORD DTMFabort; /* Abort on DTM */ INT32 gain; /* Recording gain in dB */ /*-[SLC parms (used if silence det)] DWORD novoicetime; /* Length of initial silence to stop /* Recording (ms); use 0 to deactivate /* Initial silence detection. DWORD silencetime; /* Length of silence to stop recording /* After voice has been detected (ms); /* Use 0 to deactivate. INT32 silenceampl; /* Qualif level for silence (dBm) WORD silencedeglitch; /* Deglitch while qualifying silence(ms) /*-[Beep for record]----- DWORD beepfreq; /* Beep frequency (Hz) INT32 beepampl; /* Beep amplitude (dBm) DWORD beepetime; /* Beep time (ms) 0=no beep /*--[AGC parms]----- WORD AGCenable; /* Enable AGC; use 1 to activate INT32 AGCtargetampl; /* Target AGC level (dBm) INT32 AGCsilenceampl; /* Silence level (dBm) DWORD AGCattacktime; /* Attack time (ms) DWORD AGCdecaytime; /* Decay time (ms) } ADI_RECORD_PARMS;</pre> <p>Note: Fields in bold are not applicable to the native play and record feature.</p> <p>Refer to <i>ADI_RECORD_PARMS</i> on page 262 for field descriptions and valid values.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_BAD_ARGUMENT	Invalid encoding or NULL buffer .
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_OUTPUT_ACTIVE	Record failed because there is another active output function.
CTAERR_RESOURCE_CONFLICT	Silence detector is in use by adiStartEnergyDetector .
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_RECORDING_DONE	<p>Generated when the recording operation terminates. The event size field contains the total number of bytes written into the buffer. The value field contains one of the following termination reasons or error codes:</p> <p>CTA_REASON_DIGIT Aborted due to DTMF.</p> <p>CTA_REASON_FINISHED Buffer filled.</p> <p>CTA_REASON_NO_VOICE No voice detected.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p> <p>CTA_REASON_VOICE_END User stopped speaking.</p> <p>CTAERR_FUNCTION_NOT_AVAIL Required DSP file not loaded on the board.</p> <p>CTAERR_xxx or ADIERR_xxx Record failed.</p>

Details

When recording or playing speech files on AG boards, a specific DSP file must be loaded for each encoding type. For QX boards, the standard DSP file supports the valid encoding types. For more information, refer to *Voice encoding formats* on page 13.

When recording or playing speech files on CG boards, a specific DSP file must be loaded for each encoding type except when using the native play and record feature. The native play and record feature combines an ADI port with an MSPP endpoint and plays or records speech data directly to or from an IP endpoint with no transcoding. For information on the native play and record feature, refer to *Performing NMS native play and record* on page 31.

For more information, see *Encoding formats and DSP files* on page 134. The table lists the DSP files that must be loaded on the AG and CG boards. It also lists the valid encoding types that QX boards and PacketMedia HMP processes support. Refer to the board installation and developer's manual for MIPS usage.

Use **adiRecordToMemory** to initiate recording to memory-resident **buffer** of size **bufsize** and return to the application. The ADI service records data into the buffer until one of the terminating conditions described in ADIEVN_RECORDING_DONE occurs.

Note: You cannot initiate a record operation while playing voice or generating tones unless you disable the record beep by setting either ADI_RECORD.beeptime or ADI_RECORD.beepfreq to 0 (zero). You cannot start a record operation if the energy detector is active, unless both ADI_RECORD.novoicetime and ADI_RECORD.silencetime are 0 (zero).

For more information, refer to *Recording* on page 20.

See also

adiCommandRecord, **adiGetEncodingInfo**, **adiSetNativeInfo**,
adiStopRecording

Example

```

/* Record to supplied buffer, stopping after 1 second of silence. */
int myRecord( CTAHD ctahd, unsigned encoding,
              void *buf, unsigned bufsize, unsigned *bytesrecorded )
{
    ADI_RECORD_PARMS parms;
    CTA_EVENT          event;
    unsigned            datarate;                /* average bytes/sec */
    int                 myret;
    unsigned            silencetime = 1000;
    unsigned            trimsize = 0;

    /* Modify default silence timeout */
    ctaGetParms (ADI_RECORD_PARMID, &parms, sizeof parms);
    parms.silencetime = silencetime;

    if( adiRecordToMemory (ctahd, encoding, buf, bufsize, &parms) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event );                /* see ctaWaitEvent example */
    } while (event.id != ADIEVN_RECORD_DONE);

    switch (event.value)
    {
        case CTA_REASON_FINISHED:                /* Buffer filled */
            myret = SUCCESS;
            break;

        case CTA_REASON_NO_VOICE:                /* No voice detected */
            *bytesrecorded = 0;
            myret = SUCCESS;
            break;

        case CTA_REASON_RELEASED:                /* The call was terminated */
            myret = MYDISCONNECT;
            break;

        case CTA_REASON_STOPPED:                /* adiStopRecording was called */
        case CTA_REASON_DIGIT:                /* Aborted due to touchtone */
            /* DTMF is trimmed automatically by AG board */
            *bytesrecorded = event.size;
            myret = SUCCESS;
            break;

        case CTA_REASON_VOICE_END:                /* Silence after voice */
            *bytesrecorded = event.size;
            adiGetEncodingInfo (ctahd, encoding, NULL, &datarate, NULL);
            trimsize = datarate * silencetime / 1000;
            myret = SUCCESS;
            break;

        default:                                /* an error code */
            myret = MYFAILURE;
            break;
    }

    if (myret == SUCCESS)
    {
        if (*bytesrecorded > trimsize)
            *bytesrecorded -= trimsize;
        else
            *bytesrecorded = 0;
    }

    return myret;
}

```

adiSetBoardClock

Sets the time on an AG or a CG board.

Supported board types

- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiSetBoardClock** (CTAHD *ctahd*, unsigned *board*, unsigned long *time*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>board</i>	Board number as specified in the board keyword file.
<i>time</i>	Number of seconds elapsed since 1/1/70.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_SEND_FAILED	Invalid board.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiSetBoardClock** to update the time on an AG board or a CG board, affecting the timestamp in all events that originate on the board. *board* does not have to be the same board that the ADI service is opened on, but it must be the same family of board (AG or CG). If you are opening the ADI service only to set the clock, set `services[0].mvipaddr.mode` to 0 (zero) in the call to **ctaOpenServices** so that no timeslot is used.

See also

adiGetTimeStamp

Example

```
int mySetBoardClock (CTAHD ctahd, unsigned board)
{
    time_t ltime = time(NULL);
    return adiSetBoardClock( ctahd, board, ltime);
}
```

adiSetNativeInfo

Enables native play and record mode for an ADI port.

Supported board types

- CG
- PacketMedia HMP process

Prototype

DWORD **adiSetNativeInfo** (CTAHD *ctahd*, DWORD *ingresshd*, DWORD *egresshd*, ADI_NATIVE_CONTROL **control*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>ingresshd</i>	MSPP filter handle that connects the ADI native record channel on the board to the MSP jitter buffer.
<i>egresshd</i>	MSPP endpoint handle that connects the ADI native play channel on the board to the RTP endpoint.

Argument	Description
control	<p>Pointer to the ADI_NATIVE_CONTROL structure, as shown:</p> <pre> typedef struct { DWORD size; /* Size of this structure */ DWORD mode; /* Enables and disables native */ /* play/record mode for the adi port. */ /* (ADI_NATIVE, ADI_IVR_ONLY) */ DWORD play_encoding; /* Encoding type for native play (refer to adidef.h*/ /* for a complete list of ADI_ENCODING_xxx /* values).*/ /* If the mode is ADI_NATIVE and this encoding /* matches the one specified in a subsequent play /* command, the native path will be used, /* otherwise*/ /* the PSTN play or record path will be used. */ DWORD rec_encoding; /* Encoding type for native record (refer to /* adidef.h for a complete list of /* ADI_ENCODING_xxx*/ /* values). If the mode is ADI_NATIVE and this /* encoding type matches the one specified in a /* subsequent record command, the native path will /* be used, otherwise the PSTN play or record path /* will be used. */ DWORD frameformat; /* specifies record frame format */ DWORD include2833; /* include RFC2833 markers in record buffers */ WORD payloadID; /* RTP payload type for egress */ WORD nsPayload; /* nonstandard payload indicator for Egress RTP, /* 0 (default)=RFC3267 AMR payload is used, /* 1=AMR IF2 frames are packed as payload) */ WORD vadFlag; /* VAD enable (1)/disable (0) sending of SID /* frames*/ } ADI_NATIVE_CONTROL; </pre> <p>Refer to the Details section for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Details

Use **adiSetNativeInfo** to set the native play and record parameters. The native play and record feature enables applications to use the ADI service to play and record voice data directly to and from RTP endpoints associated with MSPP service connections. For information about the native play and record feature, refer to *Performing NMS native play and record* on page 31.

To enable the native play and record feature in the `ADI_NATIVE_CONTROL` structure, set the mode to `ADI_NATIVE`. Also set the encoding type so that it matches the encoding type specified in the associated ADI play or record function calls. The specified encoding type must be one of the `ADI_ENCODE_EDTX` formats. For information about encoding formats, refer to *Recording and playing* on page 13.

Subsequent play calls specifying an encoding type with the same base codec type use the native path to play directly to the MSPP filter. The egress handle in this function specifies the MSPP filter. For example, if the `ADI_ENCODE_EDTX_G723` is specified in the call, subsequent play or record calls specifying `ADI_ENCODE_G723_6`, `ADI_ENCODE_G723_5`, `ADI_ENCODE_EDTX_G723`, `ADI_ENCODE_EDTX_G723_6`, or `ADI_ENCODE_EDTX_G723_5` use the native play path.

Subsequent record calls specifying an `ADI_ENCODE_EDTX` encoding type with the same base codec type use the native path to record from the MSPP filters. The ingress handle in this function specifies the MSPP filters. For example, if the `ADI_ENCODE_EDTX_G723` is specified in the call, subsequent play or record calls specifying `ADI_ENCODE_EDTX_G723`, `ADI_ENCODE_EDTX_G723_6`, or `ADI_ENCODE_EDTX_G723_5` use the native record path.

To disable the native feature in the `ADI_NATIVE_CONTROL` structure, set the mode to `ADI_IVR_ONLY`.

Because the native record mode responds to silence as well as to audio data, the ADI port requires DSP resources for silence detection.

The ADI_NATIVE_CONTROL structure contains the following fields:

Field	Description
size	Size of the structure in bytes.
mode	ADI_NATIVE = use the ADI service to play and record voice data directly to and from RTP endpoints associated with MSPP service connections. ADI_IVR_ONLY = use the PSTN play or record path.
play_encoding	Native play data encoding format. See <i>Recording and playing</i> on page 13 for a complete list.
rec_encoding	Native record data encoding format. See <i>Recording and playing</i> for a complete list.
frameformat	Frame format in record buffers: 0 = Variable frame size with compressed silence 1 = Variable frame size with expanded silence 2 = Fixed frame size with compressed silence 3 = Fixed frame size with expanded silence
include2833	RFC2833 markers in record buffers: 0 = Disable 1 = Enable
payloadID	Payload type used in egress RTP packets as defined in RFC3551.
nsPayload	Payload format of egress RTP packets: 0 = Standard payload format 1 = Nonstandard payload format
vadFlag	Send SID frames: 0 = Enable (default) 1 = VAD disable

See also

adiPlayAsync, adiPlayFromMemory, adiRecordAsync, adiRecordToMemory, adiStartPlaying, adiStartRecording

Example

```
void mySetNativeInfo( CTAHD ctahd, DWORD ingresshd, DWORD egresshd, int encoding, int
payloadID)
{
    ADI_NATIVE_CONTROL np;
    np.rec_encoding = encoding;
    np.play_encoding = encoding;
    np.frameformat = 0;
    np.include2833 = 0;
    np.mode = ADI_NATIVE;
    np.nsPayload = 0;
    np.payloadID = payloadID;
    np.vadFlag = 1
    if((ret =adiSetNativeInfo( ctahd, ingresshd, egresshd, &np)) == SUCCESS )
        printf( "Set Native Control Successful  for handle %x\n", ctahd );
    else
        printf( "Set Native Control Failed (%x)  for handle %x\n",ret, ctahd );
    return;
}
```

adiStartCallProgress

Starts monitoring call progress analysis data.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStartCallProgress** (CTAHD *ctahd*, ADI_CALLPROG_PARMS **parms*)

Argument	Description
ctahd	Context handle returned by ctaCreateContext or ctaAttachContext .
parms	<p>Pointer to call progress analysis parameters, stored in the ADI_CALLPROG_PARMS, as follows (NULL designates default values):</p> <pre>typedef struct { DWORD size; // Size of this structure DWORD timeout; // If no tone/voice detected, done via timeout (ms) DWORD busycount; // Number of busy cycles until report and quit; // busycount ignored if precise busy detected. DWORD ringcount; // Number of ring cycles until report and quit. DWORD maxreorder; // Separates fast busy from busy (ms) DWORD maxbusy; // Separates busy from ring cycle (ms) DWORD maxring; // Separates ring from dial tones (ms) DWORD maxringperiod; // Maximum ring period before CP_RING_QUIT (ms) DWORD voicemedium; // Time after VOICE BEGIN until VOICE MEDIUM (ms) DWORD voicelong; // Time after VOICE BEGIN until VOICE LONG (ms) DWORD voicextended; // Time after VOICE BEGIN until VOICE EXTENDED (ms) DWORD silencetime; // Silence period after voice til VOICE END (ms) DWORD precqualtime; // Precise tone qualification time (ms) DWORD precmask; // Precise tone mask DWORD stopmask; // mask to auto-stop adiCallProgress: INT32 silencelevel; // Reference level below which is "silence" (dBm) DWORD voicetoneration; // voice vs. tone ratio (IDUs) DWORD qualtonetime1; // Qualify time 1 for the TONE state (ms); DWORD qualtonetime2; // Qualify time 2 for the TONE state (ms); DWORD qualvoicetime1; // Qualify time 1 for the VOICE state (ms); DWORD qualvoicetime2; // Qualify time 2 for the VOICE state (ms); DWORD leakagetime; // Leaky integrator time constant (in ms) DWORD noiselevel; // Level window for QT2 state (in IDUs) } ADI_CALLPROG_PARMS;</pre> <p>Refer to <i>ADI_CALLPROG_PARMS</i> on page 254 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument had an invalid value, or a required pointer argument was NULL.
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_CP_BUSYTONE	Detected a busy tone.
ADIEVN_CP_CED	Detected a called party modem or fax terminal tone.
ADIEVN_CP_DIALTONE	Detected a dial tone.
ADIEVN_CP_DONE	Call progress analysis terminated normally. The event value field can contain one of the following termination conditions or an error code: CTA_REASON_FINISHED CTA_REASON_TIMEOUT CTA_REASON_RELEASED CTA_REASON_STOPPED
ADIEVN_CP_NOANSWER	Detected no answer. The parameterized number of rings were detected without voice being detected.
ADIEVN_CP_RINGQUIT	Call progress analysis stopped detecting ring tones. Ring was previously detected and another ring was not detected in time. The cause can be a network error or a soft speaker answering the phone.
ADIEVN_CP_RINGTONE	Detected ring tone.
ADIEVN_CP_REORDERTONE	Detected a reorder (fast-busy) tone.
ADIEVN_CP_SIT	Detected a special information tone (SIT). If ADI_CPMSK_PRECISE_SITEXT is set in precmask, the low order three bits in the event value field indicate the type of SIT detected: 001 = intercept 011 = reorder; ineffective other 101 = vacant code 111 = no circuit available
ADIEVN_CP_STOPPED	Call progress analysis terminated by the application.

Event	Description
ADIEVN_CP_TDD	Detected a TDD/TTY device tone.
ADIEVN_CP_VOICE	Call progress analysis detected voice. The event value field contains one of the following: ADI_CP_VOICE_BEGIN ADI_CP_VOICE_MEDIUM ADI_CP_VOICE_LONG ADI_CP_VOICE_EXTENDED ADI_CP_VOICE_END

Details

The following DSP files must be loaded to the board before running **adiStartCallProgress**:

For these boards...	Load these DSP files...
AG	<i>callp.m54</i> <i>ptf.m54</i>
CG	<i>callp.f54</i> <i>ptf.f54</i>
QX	The standard QX DSP file

Refer to *DSP file summary* on page 269 for DSP descriptions. Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Use this function to start the call progress analysis operation. This is the same functionality utilized by call control. It can be used by applications that are not using standard call control, or by any application during the connected state.

Caution:	Modifying the following fields in ADI_CALLPROG_PARMS can compromise your application's ability to interact with the telephone network: voicetonratio qualtonetime1 qualtonetime2 qualvoicetime1 qualvoicetime2 leakagetime noiselevel
-----------------	--

The call progress analysis operation always terminates when any of the following events occurs:

- ADIEVN_CP_DIALTONE
- ADIEVN_CP_BUSYTONE
- ADIEVN_CP_RORDTONE
- ADIEVN_CP_SIT
- ADIEVN_CP_NOANSWER
- ADIEVN_CP_CED
- ADIEVN_CP_TDD

You can configure the ADI_CALLPROG_PARMS stopmask parameter to stop when any of the following events occur:

- ADIEVN_CP_RINGTONE
- ADIEVN_CP_RINGQUIT
- ADIEVN_CP_VOICE_BEGIN
- ADIEVN_CP_VOICE_MEDIUM
- ADIEVN_CP_VOICE_LONG
- ADIEVN_CP_VOICE_EXTENDED
- ADIEVN_CP_VOICE_END

See also

adiStopCallProgress

Example

```

/* Wait for voice detection or any network tone.
 * Returns SUCCESS if voice is detected within 30 seconds, else DISCONNECT.
 */
int waitforvoice( CTAHD ctahd )
{
    ADI_CALLPROG_PARMS    parms;
    CTA_EVENT              event;
    DWORD                  last_cp_event = 0;

    ctaGetParms( ctahd, ADI_CALLPROG_PARMID, &parms, sizeof parms);
    parms.stopmask |= ADI_CPSTOP_ON_VOICE_BEGIN;
    parms.timeout = 30000; /* Increase timeout from default 10 seconds */

    if( adiStartCallProgress (ctahd, &parms) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event ); /* see ctaWaitEvent example */

        if (ADIEVN_CP_VOICE <= event.id && event.id <= ADIEVN_CP_CED)
            last_cp_event = event.id;

    } while (event.id != ADIEVN_CP_DONE);

    switch (event.value)
    {
        case CTA_REASON_FINISHED:
            if (last_cp_event == ADIEVN_CP_VOICE)
                return SUCCESS;
            else
                return MYDISCONNECT; /* hang-up tone detected */

        case CTA_REASON_TIMEOUT: /* nothing detected - give up */
        case CTA_REASON_RELEASED: /* The call was terminated */
        case CTA_REASON_STOPPED:
        default:
            return MYDISCONNECT;
    }
}

```

adiStartDial

Starts the dialing function for applications that are not using protocol-independent call control.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartDial** (CTAHD *ctahd*, char **digitstr*, ADI_DIAL_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>digitstr</i>	Pointer to string of digits to be dialed (ADI_MAX_DIGITS).
<i>parms</i>	<p>Pointer to dialing parameters, stored in ADI_DIAL_PARMS structure as follows (NULL designates default values):</p> <pre> typedef struct { WORD size; /* size of this structure */ DWORD method; /* default dialing method: 0=DTMF, 1=pulse, 2=MF */ DWORD breaktime; /* duration of pulse digit break (ms) */ DWORD maketime; /* duration of pulse digit make (ms) */ DWORD interpulse; /* interdigit delay between pulsed digits (ms) */ DWORD flashtime; /* duration of the flash-hook (ms) */ DWORD shortpause; /* duration of the comma in dialing string (ms) */ DWORD longpause; /* duration of the dot in dialing string (ms) */ INT32 dtmfamp1; /* first dtmf amplitude (dBm) */ INT32 dtmfamp2; /* second dtmf amplitude (dBm) */ DWORD dtmfontime; /* ON duration of DTMFs (ms) */ DWORD dtmfofftime; /* OFF duration of DTMFs (ms) */ DWORD dialtonewait; /* max time to wait for dialtone (ms) on ';' */ /* precise dialtone parameters: */ DWORD tonefreq1; /* frequency to detect */ DWORD tonebandw1; /* bandwidth */ DWORD tonefreq2; /* 2nd frequency to detect (dualtone) */ DWORD tonebandw2; /* 2nd bandwidth */ INT32 tonequalamp1; /* broadband qual level (in dBm) */ DWORD tonequaltime; /* qualification time (in ms) */ DWORD tonereflevel; /* reserved */ DWORD reserved; /* reserved, must be 0 */ DWORD tonetotaltime; /* total time for dial tone with interruptions */ } ADI_DIAL_PARMS; </pre> <p>In some instances, the dtmfofftime can increase by 20 ms.</p> <p>Refer to <i>ADI_DIAL_PARMS</i> on page 257 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	digitstr is NULL.
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_OUTPUT_ACTIVE	Open port failed because the stream and slot are already opened on another port.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_DIAL_DONE	The reason in the value field can contain either an error or one of the following: CTA_REASON_FINISHED CTA_REASON_RELEASED CTA_REASON_STOPPED

Details

For AG boards and CG boards, **adiStartDial** requires one or more of the following DSP files to be loaded, depending on the **digitstr** and related parameters:

CG boards	AG boards	Use
<i>tone.f54</i>	<i>tone.m54</i>	Generating DTMF and MF tones.
None	<i>signal.m54</i> (AG 2000, AG 2000C, and AG 2000-BRI only)	Generating pulse digits.
<i>ptf.f54</i>	<i>ptf.m54</i>	Precise dial tone detection.

For QX boards, this function is supported in the standard DSP file. Refer to the *QX 2000 Installation and Developer's Manual* for a table of MIPS usage for all functions.

Use this function to start dialing for NOCC users. **adiStartDial** is similar to **adiStartDTMF**, but also allows access to pulse-dialing, flashing, and precise dial tone detection.

Note: The DTMF detector is disabled while **adiStartDial** is active.

digitstr can contain the following embedded escape characters that control the dialing sequence:

Character	Description
;	(semicolon) Wait for precise dial tone.
.	(period) Insert long pause in dialing.
,	(comma) Insert short pause in dialing.
!	Flash hook.
P	Switch to pulse dialing.
T	Switch to DTMF dialing.
M	Switch to MF dialing.

The following table lists the mapping to the United States MF digits for MF dialing:

Digit	United States MF name
0 to 9	Specific digit address
B	MF ST3P
C	MF STP
D	MF KP
E	MF KP2, MF ST2P
F	MF ST

NOCC users can start a call progress analysis operation (**adiStartCallProgress**) after receiving the DONE event.

After calling **adiStartDial**, expect a DONE event. If the function is dialing DTMFs, the outbound voice path must be available (not in use by another function). If the function is pulse dialing, the outbound signaling path must be available.

See also

adiStartMFDetector, adiStartProtocol, adiStopDial

adiStartDTMF

Starts generating a string of DTMFs or MFs.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartDTMF** (CTAHD *ctahd*, char **digits*, ADI_DTMF_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>digits</i>	Pointer to a string of DTMF digits including 0 though 9, A through F, * (asterisk), # (number sign), and , (comma) or . (period) for pauses. All other characters are ignored.
<i>parms</i>	<p>Pointer to DTMF parameters according to the following structure (NULL value uses the default values):</p> <pre>typedef struct { DWORD size; /* size of this structure */ INT32 ampl1; /* level of first tone (dBm) */ INT32 ampl2; /* level of second tone (dBm) */ DWORD ontime; /* on duration of DTMF tone (ms) */ DWORD offtime; /* off duration of DTMF tone (ms) */ DWORD shortpause; /* duration of ',' (ms) */ DWORD longpause; /* duration of '.' (ms) */ } ADI_DTMF_PARMS;</pre> <p>In some instances, the dtmfofftime can increase by 20 ms.</p> <p>Refer to <i>ADI_DTMF_PARMS</i> on page 259 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_OUTPUT_ACTIVE	Open port failed because the stream and slot are already opened on another port.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_TONES_DONE	Value field can contain CTA_REASON_FINISHED or CTA_REASON_STOPPED.

Details

The following DSP file must be loaded to the board before running **adiStartDTMF**:

For these boards...	Load this DSP file...
AG	<i>tone.m54</i>
CG	<i>tone.f54</i>
QX	The standard QX DSP file

See *DSP file summary* on page 269 for DSP file descriptions. Refer to the board-specific installation and developer's manual for a table of MIPS usage for all functions.

Use this function to start generating a sequence of DTMF tones or MF tones. Use **adiStopTones** to terminate DTMF or MF generation.

Note: While **adiStartDTMF** is active, the DTMF detector is disabled.

digits can contain the following embedded escape characters that control the dialing sequence:

Character	Description
.	(period) Insert long pause in dialing.
,	(comma) Insert short pause in dialing.
T	Switch to DTMF dialing (default).
M	Switch to MF dialing.

To generate MF tones, precede the string with an M.

The following table lists the mapping to the United States MF digits for MF dialing:

Digit	United States MF name
0 to 9	Specific digit address
B	MF ST3P
C	MF STP
D	MF KP
E	MF KP2, MF ST2P
F	MF ST

See also

adiStartTones

adiStartDTMFDetector

Starts DTMF detection.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartDTMFDetector** (CTAHD *ctahd*, ADI_DTMFDETECT_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>parms</i>	<p>Pointer to DTMF detection parameters according to the following structure (NULL value uses the default values):</p> <pre>typedef struct { DWORD size; /* size of this structure */ DWORD columnfour; /* 1=detect DTMFs A,B,C,D; 0=don't */ INT32 onqualampl; /* min input lev to qual tone (dBm) */ DWORD onthreshold; /* reserved */ DWORD onqualtime; /* qualify time of DTMF (ms) */ INT32 offqualampl; /* min input lev of valid DTMF (dBm)*/ DWORD offthreshold; /* reserved */ DWORD offqualtime; /* disqualify time for tone (ms) */ } ADI_DTMFDETECT_PARMS;</pre> <p>Refer to <i>ADI_DTMFDETECT_PARMS</i> on page 260 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_DIGIT_BEGIN	Raw DTMF digit detected on.
ADIEVN_DIGIT_END	Raw DTMF digit detected off.
ADIEVN_DTMF_DETECT_DONE	DTMF detector no longer running. The event value field contains one of the following: CTA_REASON_RELEASED Call terminated. CTAERR_xxx or ADIERR_xxx DTMF detector failed. CTA_REASON_STOPPED Function stopped with adiStopDTMFDetector .

Details

The following DSP file must be loaded to the board before running **adiStartDTMFDetector**:

For these boards...	Load this DSP file...
AG	<i>dtmf.m54</i> or <i>dtmfe.m54</i>
CG	<i>dtmf.f54</i> or <i>dtmfe.f54</i>
QX	The standard QX DSP file

See *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Use this function to enable detection of DTMFs. By default, the DTMF detector is active after the NOCC protocol is started, or when the context enters the call control connected state. Therefore, this function is needed only to restart the DTMF detector if it was stopped using **adiStopDTMFDetector** or to modify the DTMF detector parameters.

An ADIEVN_DIGIT_BEGIN event and an ADIEVN_DIGIT_END event are generated for every digit detected. Each event contains the ASCII representation (character) of the digit: 0 through 9, A through D, * (asterisk), or # (number sign) in the event value field.

The DTMF detector must be enabled for the digit collection functions (for example, **adiCollectDigits** or **adiGetDigit**).

adiStartEnergyDetector

Starts the energy detector.

Note: Do not use the energy detector if you are using voice activity detection.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartEnergyDetector** (CTAHD *ctahd*, unsigned *energyqual*, unsigned *silencequal*, ADI_ENERGY_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>energyqual</i>	Qualification time for energy (in milliseconds).
<i>silencequal</i>	Qualification time for silence (in milliseconds).
<i>parms</i>	<p>Pointer to energy detector parameters according to the following structure (NULL designates default values):</p> <pre>typedef struct { DWORD size; /* parameters for energy detection:*/ /* size of this structure */ INT32 thresholdamp1; /* silence level (dBm) */ DWORD deglitch; /* ms deglitch during transitions */ DWORD autostop; /* on detection, 1=autostop 0=don't*/ } ADI_ENERGY_PARMS;</pre> <p>Refer to <i>ADI_ENERGY_PARMS</i> on page 260 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_RESOURCE_CONFLICT	Silence detector in use by a record function.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_ENERGY_DETECTED	Energy detector reporting energy.
ADIEVN_ENERGY_DETECT_DONE	Energy detector terminated.
ADIEVN_SILENCE_DETECTED	Energy detector reporting silence.

Details

The following DSP file must be loaded to the board before running **adiStartEnergyDetector**:

For these boards...	Load this DSP file...
AG	<i>dtmf.m54</i> or <i>dtmfe.m54</i>
CG	<i>dtmf.f54</i> or <i>dtmfe.f54</i>
QX	The standard QX DSP file

See *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Use this function to start a low-level energy detector that reports energy and silence transitions.

The `thresholdampl` is the dBm threshold below which is considered silence. Once energy or silence is internally qualified as detected, the `deglitch` time is used during transitions above and below the threshold. The `autostop` field indicates that the function stops once energy or silence is detected. The maximum valid value for `energyqual` and `silencequal` is 65535.

If `autostop` is set, `ADIEVN_ENERGY_DETECT_DONE` is received with the value field set to `CTA_REASON_FINISHED` and the size field is set to either `ADIEVN_ENERGY_DETECTED` or `ADIEVN_SILENCE_DETECTED`.

In continuous mode, `ADIEVN_ENERGY_DETECTED` and `ADIEVN_SILENCE_DETECTED` are received as the detector changes between these states.

`ADIEVN_ENERGY_DETECT_DONE` can also be returned with the value field set to an error or `CTA_REASON_STOPPED` if **adiStopEnergyDetector** is called.

Note: You cannot start the energy detector while a record operation is active unless both `ADI_RECORD.novoicetime` and `ADI_RECORD.silencetime` are 0 (zero) when the record operation was started. For a voice record operation, the relevant parameters are `VCE_RECORD.novoicetime` and `VCE_RECORD.silencetime`.

For more information, refer to *Detecting energy* on page 66.

adiStartMFDetector

Enables the detection of MFs (multi-frequency tones).

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStartMFDetector** (CTAHD *ctahd*, unsigned *mftype*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>mftype</i>	Type of MF tone to detect: ADI_MF_US ADI_MF_CCITT_FORWARD ADI_MF_CCITT_BACKWARD

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_FUNCTION_NOT_AVAIL	<i>mf.dsp</i> not loaded to the board.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_MF_DETECT_DONE	DTMF detector is no longer running. The event value field contains one of the following: CTA_REASON_RELEASED Call terminated. CTA_REASON_STOPPED Function stopped by adiStopMFDetector . CTAERR_XXX or ADIERR_XXX DTMF detector failed.
ADIEVN_MF_DIGIT_BEGIN	MF digit detected on.
ADIEVN_MF_DIGIT_END	MF digit detected off.

Details

The following DSP file must be loaded to the board before running **adiStartMFDetector**:

For these boards...	Load this DSP file...
AG	<i>mf.m54</i>
CG	<i>mf.f54</i>
QX	The standard QX DSP file

See *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Use this function to enable detection of MFs (multi-frequency tones). Stop the function by calling **adiStopMFDetector**.

You must disable DTMF detection using **adiStopDTMFDetector** before initiating MF detection, as there are some overlapping frequency ranges in which both a DTMF and an MF event are reported. Likewise, when the application is finished with MF detection, re-enable DTMF detection (**adiStartMFDetector**) if DTMFs are desired.

ADIEVN_MF_DIGIT_BEGIN and ADIEVN_MF_DIGIT_END are generated for every MF tone detected. Each event contains an ASCII representation (character) of the MF digit in the event value field.

This table lists the MF frequencies for each *mftype*, along with the digit value returned in the event's value field:

Digit	US MF	ITU forward	ITU backward
1	700,900	1380,1500	1140,1020
2	700,1100	1380,1620	1140,900
3	900,1100	1500,1620	1020,900
4	700,1300	1380,1740	1140,780
5	900,1300	1500,1740	1020,780
6	1100,1300	1620,1740	900,780
7	700,1500	1380,1860	1140,660
8	900,1500	1500,1860	1020,660
9	1100,1500	1620,1860	900,660
0	1300,1500	1740,1860	780,660
B	700,1700	1380,1980	1140,540
C	900,1700	1500,1980	1020,540
D	1100,1700	1620,1980	900,540
E	1300,1700	1740,1980	780,540
F	1500,1700	1860,1980	660,540

The following table lists the mapping to the United States MF digits for MF dialing:

Digit	United States MF name
0 to 9	Specific digit address
B	MF ST3P
C	MF STP
D	MF KP
E	MF KP2, MF ST2P
F	MF ST

Note: The digit values are the same as those used by **adiStartDTMF** and **adiStartDial** when MF digits are dialed.

See also

adiStartDTMFDetector, **adiStopDTMFDetector**

adiStartPlaying

Starts a playing operation using a callback routine to get data. This function is not supported when Natural Access is running in client/server mode.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartPlaying** (CTAHD *ctahd*, unsigned *encoding*, ADIPLAY_ACCESS *access*, void **userarg*, ADI_PLAY_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>encoding</i>	Data encoding selection. See <i>Voice encoding formats</i> on page 13 for a complete list.
<i>access</i>	Pointer to a callback function that supplies data to be played. See the Details section for a prototype of this function.
<i>userarg</i>	An arbitrary pointer or value to be passed to the callback function every time it is invoked.
<i>parms</i>	<p>Pointer to play parameters according to the following structure (NULL value uses default values):</p> <pre>typedef struct { DWORD size; /* parms related to adiStartPlaying: */ /* size of this structure */ DWORD DTMFabort; /* abort on DTMF */ INT32 gain; /* playing gain in dB */ DWORD speed; /* initial speed in percent */ DWORD maxspeed; /* maximum play speed in percent */ } ADI_PLAY_PARMS;</pre> <p>Refer to <i>ADI_PLAY_PARMS</i> on page 261 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_BAD_ARGUMENT	Invalid encoding .
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_NO_MEMORY	Could not allocate an internal buffer.
CTAERR_NOT_IMPLEMENTED	Function not implemented.
CTAERR_OUTPUT_ACTIVE	Play failed because there is another active output function.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_PLAY_DONE	<p>Playing terminated, with one of the following reasons in the value field:</p> <p>CTAERR_*** or ADIERR_*** Error codes indicate play failed.</p> <p>CTA_REASON_DIGIT Aborted due to DTMF.</p> <p>CTA_REASON_FINISHED ADI service finished playing the last buffer.</p> <p>CTA_REASON_RECOGNITION Aborted because of speech recognition.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p>

Details

When recording or playing speech files on AG boards, a specific DSP file must be loaded for each encoding type. For QX boards, the standard DSP file supports the valid encoding types. For more information, refer to *Voice encoding formats* on page 13.

When recording or playing speech files on CG boards, a specific DSP file must be loaded for each encoding type except when using the native play and record feature. The native play and record feature combines an ADI port with an MSPP endpoint and plays or records speech data directly to or from an IP endpoint with no transcoding. For information about the native play and record feature, refer to *Performing NMS native play and record* on page 31.

For more information, see *Encoding formats and DSP files* on page 134. The table lists the DSP files that must be loaded on AG and CG boards. It also lists the valid encoding types that QX boards and PacketMedia HMP processes support. Refer to the board installation and developer's manual for MIPS usage.

For QX boards and the PacketMedia HMP process, the maxspeed and speed parameters are not used.

The ADI service allocates a buffer and invokes the **access** function provided by the programmer. The buffer and size are passed to the callback function and the application must fill the buffer with voice data (for example, read data from a file) before returning.

access is invoked from within **adiStartPlaying** for the first buffer and subsequently invoked from within **ctaWaitEvent**. The prototype for the callback function is:

```
int NMSSTDCALL access ( void *userarg, void *buffer, unsigned size, unsigned *rsize )
```

Argument	Description
<i>userarg</i>	Pointer to value previously passed to adiStartPlaying .
<i>buffer</i>	Pointer to memory to be filled with voice data.
<i>size</i>	Size (bytes) of the buffer.
<i>rsize</i>	Returned number of bytes of voice data put into the buffer by the callback routine. This value is returned to the ADI service.

access has the following return values:

Return value	Description
SUCCESS	Play continues as normal. The ADI service invokes access again when needed.
ADI_PLAY_LAST_BUFFER	When the ADI service finishes playing the buffer being returned from access , ADIEVN_PLAY_DONE is generated with the value field set to CTA_REASON_FINISHED. access is not invoked again for the current playing instance.

If the **access** return value is neither SUCCESS nor ADI_PLAY_LAST_BUFFER, the ADI service immediately terminates the playing instance. ADIEVN_PLAY_DONE is generated and the value field is set to ADIERR_PLAYREC_ACCESS.

access returns the number of bytes written to the buffer in the *rsize* variable. If the returned size is larger than the buffer or the returned size is not a multiple of the **framesize** for the given encoding, the ADI service terminates the play function and generates ADIEVEN_PLAY_DONE and the value field is set to CTAERR_BAD_SIZE.

Note: Starting a play operation with the maxspeed parameter greater than 100 consumes additional DSP cycles. You may not be able to run the number of ports normally supported. Refer to the board installation and developer's manual for more information.

For more information, refer to *Playing* on page 25.

See also

adiGetPlayStatus, adiModifyPlayGain, adiModifyPlaySpeed, adiPlayAsync, adiPlayFromMemory, adiSetNativeInfo, adiStopPlaying

Example

This example shows a fragment of a program that plays the file *test.vce* using **adiStartPlaying** and the associated **access** routine.

```
int NMSSTDCALL readAccess (
    void      *userarg,
    void      *buffer,
    unsigned   size,
    unsigned  *rsize )
{
    FILE *fp = (FILE *)userarg;

    *rsize = fread( buffer, 1, size, fp );

    if ( ferror( fp ) )
        return -1;
    if ( feof( fp ) )
        return ADI_PLAY_LAST_BUFFER;
    return SUCCESS;
}

int myPlayFile( CTAHD ctahd, unsigned encoding, char *filename )
{
    CTA_EVENT event;
    FILE *fp;

    /* note: binary open */
    if ( (fp = fopen( filename, "rb" )) == NULL )
        return MYFAILURE;

    if( adiStartPlaying( ctahd, encoding, readAccess, fp, NULL ) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event ); /* see ctaWaitEvent example */
    } while( event.id != ADIEVN_PLAY_DONE );

    if( event.value == CTA_REASON_RELEASED )
        return MYDISCONNECT; /* call has been terminated */
    else if( CTA_IS_ERROR( event.value ) )
        return MYFAILURE; /* API error */
    else
        return SUCCESS; /* stopped normally */
}
```

adiStartProtocol

Starts the NOCC protocol on a specified context.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartProtocol** (CTAHD *ctahd*, char **protoname*, WORD **protoparms*, ADI_START_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>protoname</i>	Name of the protocol trunk control program (TCP). The valid value is NOCC.
<i>protoparms</i>	Valid value is NULL.
<i>parms</i>	<p>Pointer to an ADI_START_PARMS structure, as shown (NULL uses default parameter values):</p> <pre>typedef struct { DWORD size; /* size of this structure */ ADI_CALLCTL_PARMS callctl; /* call control parms */ ADI_DIAL_PARMS dial; /* dial control parms */ ADI_DTMFDETECT_PARMS dtmfdet; /* DTMF detection parms */ ADI_CLEARDOWN_PARMS cleardown; /* cleardown detect. parms*/ ADI_ECHOCANCEL_PARMS echocancel; /* echo canceller parms */ } ADI_START_PARMS;</pre> <p>Refer to <i>ADI_START_PARMS</i> on page 264 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>protoname</i> is invalid or NULL or <i>parms</i> contains an invalid size field.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_STARTPROTOCOL_DONE	<p>If successful, the value field contains CTA_REASON_FINISHED; otherwise, the value contains an error code, such as:</p> <p>CTAERR_BAD_ARGUMENT Invalid protocol name; the protocol associated with <i>protoname</i> was not specified in the configuration file.</p>

Details

Use **adiStartProtocol** to specify the NOCC protocol. The function initializes the ADI service and by default starts the DTMF detector.

The ADI_START_PARMS data structure consists of the following substructures:

- ADI_CALLCTL_PARMS controls which functions are started automatically by **adiStartProtocol**.

The ADI_CALLCTL_PARMS structure is defined as:

```
typedef struct
{
    /* call control parameters: */
    DWORD size; /* size of this structure */
    DWORD eventmask; /* not used */
    DWORD mediamask; /* functions to run: */
    #define ADI_CC_RESVDTMF 0x0001 /* reserve dtmf detection */
    #define ADI_CC_RESVSILENCE 0x0002 /* reserve silence detector */
    #define ADI_CC_RESVCLRDWN 0x0004 /* reserve clear-down det. */
    #define ADI_CC_AUTODTMF 0x0008 /* start DTMF detection */
    #define ADI_CC_AUTOECHO 0x0010 /* start echo canceller */
    #define ADI_CC_ALLMEDIA (ADI_CC_RESVDTMF|\
        ADI_CC_RESVSILENCE|ADI_CC_RESVCLRDWN|\
        ADI_CC_AUTODTMF|ADI_CC_AUTOECHO)
    DWORD blockmode; /* not used */
    DWORD debugmask; /* not used */
} ADI_CALLCTL_PARMS;
```

- ADI_DIAL_PARMS specifies how to perform dialing. Refer to **adiStartDial** for the structure definition.
- ADI_DTMFDETECT_PARMS controls DTMF detection if required by the protocol, as well as initial DTMF detection in the conversation (connected) state if started automatically by the protocol. Refer to **adiStartDTMFDetector** for the structure definition.
- ADI_ECHOCANCEL_PARMS controls the application of an echo cancellation algorithm to the context in the connected state and is defined as:

```
typedef struct
{
    /* parameters for echo cancellation*/
    DWORD size; /* size of this structure */
    DWORD mode; /* echo canceller mode */
    DWORD filterlength; /* filter length (msec) */
    DWORD adapttime; /* filter adaptation time (msec) */
    DWORD predelay; /* offset of input sample (msec) */
    INT32 gain; /* receive gain (db) */
} ADI_ECHOCANCEL_PARMS ;
```

Refer to the Parameters section for default values and a more detailed explanation of the fields in these structures.

When the protocol is NOCC, **adiStartProtocol** must be called before any ADI functions are invoked. The application can execute any function once ADIEVN_STARTPROTOCOL_DONE is received.

For details about using telephony protocols in the application, refer to the *NMS CAS for Natural Call Control Developer's Manual*.

See also**adiStopProtocol****Example**

```
int myStartProtocol( CTAHD ctahd )
{
    CTA_EVENT event;

    /* start "no call control" protocol with all default parameters */
    if( adiStartProtocol( ctahd, "NOCC", NULL, NULL ) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event );          /* see ctaWaitEvent example */
    } while( event.id != ADIEVN_STARTPROTOCOL_DONE );

    if( CTA_IS_ERROR( event.value ) )
        return MYFAILURE;              /* API error */
    else
        return SUCCESS;                /* started successfully */
}
```


adiStartPulse

Starts the generation of an out-of-band pulse.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStartPulse** (CTAHD *ctahd*, unsigned *signal*, unsigned *timeon*, unsigned *timeoff*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>signal</i>	Bit mask/pattern to pulse (assert temporarily), which is a combination of the following constants: ADI_A_BIT (0x8) ADI_B_BIT (0x4) ADI_C_BIT (0x2) ADI_D_BIT (0x1) zero (0)
<i>timeon</i>	Duration of the pulse (in milliseconds) with the pattern activated.
<i>timeoff</i>	Duration after the pulse (in milliseconds), before the DONE event is sent.

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_FUNCTION_NOT_AVAIL	Necessary .dsp file was not downloaded to the board.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_PULSE_DONE	Generated by the ADI service when the pulse function terminates. The event value field contains the termination reason.

Details

AG 2000 and AG 2000C boards require *signal.m54* to be loaded.

For QX boards, this function is supported in the standard DSP file.

Use **adiStartPulse** to output a specified signaling bit pattern for a precise duration. This function is non-blocking and returns back to the application immediately after starting the pulse.

The out-of-band **signal** pattern is either the physical out-of-band signal bits of a digital protocol or it relates to the control of an analog interface board. In both cases, four signaling bits, A, B, C, and D, often written as ABCD, and defined by a bit mask (0x8, 0x4, 0x2, and 0x1, respectively), are used. The following constants are in *adidef.h*. They can be combined by using the OR operation to define any group of bits: ADI_A_BIT, ADI_B_BIT, ADI_C_BIT, and ADI_D_BIT. For example, if the line is off-hook, a 0 (zero) is pulsed (for example, generate a flash hook).

When using this function with an analog interface board, refer to the hardware installation manual for the analog interface board for specific information on how the A and B bits affect the telephone line.

This function is not available if the current protocol reserves use of out-of-band signaling. Typically, call control protocols take over the line signaling and the application does not need to assert or reset line codes or pulses explicitly.

adiStartPulse overrides **adiAssertSignal**. For the duration of the pulse, the line pattern is determined by the signaling state specified by **adiStartPulse**. It then reverts to the pattern previously asserted by **adiAssertSignal**.

For more information, refer to *Performing low-level call control* on page 71.

adiStartReceivingFSK

Receives frequency shift key (FSK) data.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStartReceivingFSK** (CTAHD *ctahd*, void **buffer*, unsigned *bufsize*, ADI_FSKRECEIVE_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>buffer</i>	Pointer to buffer to hold received data.
<i>bufsize</i>	Size of buffer to receive.
<i>parms</i>	<p>Pointer to the FSK receive parameters, stored in the following structure (NULL designates default values):</p> <pre>typedef struct { DWORD size; /* Size of this structure */ INT32 minlevel; /* Required minimum receive level (dB) */ DWORD minmark; /* Minimum required initial mark and seizure */ DWORD droptime; /* Minimum dropout to silence before a packet is considered terminated (ms) */ DWORD baudrate; /* Baud rate (only 1200 supported) */ } ADI_FSKRECEIVE_PARMS;</pre> <p>Refer to <i>ADI_FSKRECEIVE_PARMS</i> on page 261 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	<i>buffer</i> is NULL or size is 0 (zero).
CTAERR_FUNCTION_ACTIVE	Function already started.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_FSK_RECEIVE_DONE	<p>Generated by the ADI service when the receive function terminates. The event value field contains one of the following termination conditions or an error code:</p> <p>ADI_REASON_DROP_IN_DATA Stopped due to drop in data.</p> <p>ADI_REASON_BAD_STOP_BIT Stopped due to data framing error. The stop bit at the end of data was space, not mark.</p> <p>CTA_REASON_FINISHED Data was received successfully.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p>

Details

The following DSP file must be loaded to the board before running **adiStartReceivingFSK**:

For these boards...	Load this DSP file...	
	Bellcore 1200/2200 Hz	V.23 1300/2100 Hz
AG	<i>adsir.m54</i>	<i>adsir_j.m54</i>
CG	<i>adsir.f54</i>	<i>adsir_j.f54</i>
QX	The standard QX DSP file	The standard QX DSP file

See *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Use this function to receive frequency shift key (FSK) data. The function can be stopped using **adiStopReceivingFSK**. When the function completes, ADIEVN_FSK_RECEIVE_DONE is generated.

If the event value field contains CTA_REASON_FINISHED or CTA_REASON_STOPPED, the size field of the event structure contains the number of bytes received. The received buffer is in the buffer field. If errors occur, the receive operation is terminated and the event value field contains either ADI_REASON_DROP_IN_DATA or ADI_REASON_BAD_STOP_BIT.

For more information, refer to *Sending and receiving FSK data* on page 69.

Example

```
#define MYRECEIVE_FAILURE (-11)
#define MYRECEIVE_STOPPED (-12)

int myReceiveFSK( CTAHD ctahd )
{
    CTA_EVENT event;
    char buffer [512];

    if( adiStartReceivingFSK( ctahd, buffer, sizeof buffer, NULL) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event );          /* see ctaWaitEvent example */
    } while( event.id != ADIEVN_FSK_RECEIVE_DONE );

    switch( event.value )
    {
        case CTA_REASON_FINISHED:
            return SUCCESS;

        case CTA_REASON_RELEASED:
            return MYDISCONNECT;

        case CTA_REASON_STOPPED:
            /* Receive was stopped by another application thread */
            return MYRECEIVE_STOPPED;

        case ADI_REASON_DROP_IN_DATA:
        case ADI_REASON_BAD_STOP_BIT:
            return MYRECEIVE_FAILURE;

        default:
            if( CTA_IS_ERROR( event.value ) )
                return MYFAILURE;
    }
    return MYFAILURE;
}
```

adiStartRecording

Starts a recording operation using a callback routine to deliver data. This function is not supported when Natural Access is running in client/server mode.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartRecording** (CTAHD **ctahd**, unsigned **encoding**, unsigned **maxtime**, ADIRECORD_ACCESS **access**, void ***userarg**, ADI_RECORD_PARMS ***parms**)

Argument	Description
ctahd	Context handle returned by ctaCreateContext or ctaAttachContext .
encoding	Encoding type. See <i>Voice encoding formats</i> on page 13 for a complete list.
maxtime	Maximum recording time (in milliseconds). Use zero for no time limit. When voice activity detection is enabled, maxtime is the maximum duration of speech recording, excluding silences.
access	Pointer to a function to receive recorded data. See the prototype in the Details section.
userarg	An arbitrary pointer, the value of which is passed to the callback function (access) on every invocation.
parms	<p>Pointer to record parameters according to the following structure (NULL uses default values):</p> <pre>typedef struct { DWORD size; /* size of this structure */ DWORD DTMFabort; /* abort on DTMF */ INT32 gain; /* recording gain in dB */ /*-[SLC parms (used if silence det)] DWORD novoicetime; /* length of initial silence to stop /* recording (ms); use 0 to deactivate /* initial silence detection. DWORD silencetime; /* length of silence to stop recording /* after voice has been detected (ms); /* use 0 to deactivate. INT32 silenceampl; /* qualif level for silence (dBm) DWORD silencedeglitch; /* deglitch while qualifying silence(ms) */ /*-[Beep for record]----- DWORD beepfreq; /* beep frequency (Hz) INT32 beepampl; /* beep amplitude (dBm) DWORD beeptime; /* beep time (ms) 0=no beep /*--[AGC parms]----- DWORD AGCenable; /* enable AGC; use 1 to activate INT32 AGCtargetampl; /* target AGC level (dBm) INT32 AGCsilenceampl; /* silence level (dBm) DWORD AGCattacktime; /* attack time (ms) DWORD AGCdecaytime; /* decay time (ms) } ADI_RECORD_PARMS;</pre> <p>Refer to ADI_RECORD_PARMS on page 262 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_BAD_ARGUMENT	Invalid encoding selected or NULL buffer pointer passed.
CTAERR_BAD_SIZE	size is less than one frame.
CTAERR_FUNCTION_ACTIVE	Record is already active or the energy detector is active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_NOT_IMPLEMENTED	Function not implemented.
CTAERR_OUTPUT_ACTIVE	Record failed because there is another active output function.
CTAERR_RESOURCE_CONFLICT	Silence detector is in use by adiStartEnergyDetector .
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_RECORD_DONE	<p>The value field contains one of the following termination reasons or error codes:</p> <p>CTA_REASON_DIGIT Aborted due to DTMF.</p> <p>CTA_REASON_NO_VOICE No voice detected.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p> <p>CTA_REASON_TIMEOUT Record time limit reached.</p> <p>CTA_REASON_VOICE_END User stopped speaking.</p> <p>CTAERR_FUNCTION_NOT_AVAIL Required DSP file not loaded on the board.</p> <p>CTAERR_xxx or ADIERR_xxx Record failed.</p>

Details

When recording or playing speech files on AG boards, a specific DSP file must be loaded for each encoding type. For QX boards, the standard DSP file supports the valid encoding types. For more information, refer to *Voice encoding formats* on page 13.

When recording or playing speech files on CG boards, a specific DSP file must be loaded for each encoding type except when using the native play and record feature. The native play and record feature combines an ADI port with an MSPP endpoint and plays or records speech data directly to or from an IP endpoint with no transcoding. For information about the native play and record feature, refer to *Performing NMS native play and record* on page 31.

For more information, see *Encoding formats and DSP files* on page 134. The table lists the DSP files that must be loaded on the AG and CG boards. It also lists the valid encoding types that QX boards and PacketMedia HMP processes support. Refer to the board installation and developer's manual for MIPS usage.

Use **adiStartRecording** to start a recording operation. **adiStartRecording** uses a callback routine (**access**) to deliver data. The ADI service allocates buffers and initiates recording. When a buffer fills with voice data, the ADI service invokes **access**, passing it the buffer address and size. The application must copy the buffer to a storage medium before returning from **access**.

access is invoked from **ctaWaitEvent**. The prototype for the **access** function is:

```
int NMSSTDCALL access ( void *userarg, void *buffer, unsigned size )
```

where:

Argument	Description
userarg	Pointer to value previously passed in adiStartRecording .
buffer	Pointer to memory allocated by the ADI service.
size	Size (bytes) of valid data in the buffer .

If the application's **access** returns a value other than SUCCESS, the ADI service terminates the record operation and generates ADIEVN_RECORD_DONE with a value field of ADIERR_PLAYREC_ACCESS.

Note: You cannot initiate a record operation while playing voice or generating tones unless you disable the record beep by setting either ADI_RECORD.beeptime or ADI_RECORD.beepfreq to 0 (zero). You cannot start a record operation if the energy detector is active unless both ADI_RECORD.novoicetime and ADI_RECORD.silencetime are 0 (zero).

For more information, refer to *Recording* on page 20. Refer to *ADI_RECORD_PARMS* on page 262 for field descriptions.

See also

adiCommandRecord, adiGetRecordStatus, adiRecordAsync, adiRecordToMemory, adiSetNativeInfo, adiStopRecording

Example

The following code fragment records into the file *test.vce* using **adiStartRecording**:

```
int NMSSTDCALL writeAccess(
    void      *userarg,
    void      *buffer,
    unsigned size )
{
    FILE *fp = (FILE *)userarg;

    fwrite( buffer, 1, size, fp );
    if ( ferror( fp ) )
        return -1;
    return SUCCESS;
}

int myRecordFile( CTAHD ctahd, unsigned encoding)
{
    CTA_EVENT event;
    FILE *fp;

    /* note: binary open */
    if( (fp = fopen( "test.vce", "wb" )) == NULL )
        return MYFAILURE;

    if( adiStartRecording( ctahd, encoding, 0,
                          writeAccess, fp, NULL ) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event ); /* see ctaWaitEvent example */
    } while( event.id != ADIEVN_RECORD_DONE );

    fclose( fp );

    if( event.value == CTA_REASON_RELEASED )
        return MYDISCONNECT; /* call has been terminated */
    else if( CTA_IS_ERROR( event.value ) )
        return MYFAILURE;    /* API error */
    else
        return SUCCESS;      /* stopped normally */
}
```

adiStartSendingFSK

Sends frequency shift key (FSK) data.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStartSendingFSK** (CTAHD **ctahd**, void ***buffer**, unsigned **bufsize**, ADI_FSKSEND_PARMS ***parms**)

Argument	Description
ctahd	Context handle returned by ctaCreateContext or ctaAttachContext .
buffer	Buffer to send.
bufsize	Size of buffer to send.
parms	<p>Pointer to FSK send parameters, as follows (NULL designates default values):</p> <pre>typedef struct { DWORD size; /* Size of this structure */ DWORD noseizureflag; /* No channel seizure when set */ INT32 level; /* Transmit output scaling (dBm) */ DWORD seizetime; /* Length of channel seizure in (ms) */ DWORD marktime; /* Length of the initial mark signal in (ms) */ DWORD baudrate; /* Baud rate (only 1200 supported) */ } ADI_FSKSEND_PARMS;</pre> <p>Refer to ADI_FSKSEND_PARMS on page 261 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument had an invalid value, or a required pointer argument was NULL.
CTAERR_FUNCTION_ACTIVE	Function already active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_FSK_SEND_DONE	<p>Generated by the ADI service when the send function terminates. The event value field contains one of the following termination conditions:</p> <p>CTAERR_XXX or ADIERR_XXX Function failed.</p> <p>CTA_REASON_FINISHED Buffer submitted was sent in its entirety.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p>

Details

The following DSP file must be loaded to the board before running **adiStartSendingFSK**:

For these boards...	Load this DSP file...	
	Bellcore 1200/2200 Hz	V.23 1300/2100 Hz
AG	<i>adsix.m54</i>	<i>adsix_j.m54</i>
CG	<i>adsix.f54</i>	<i>adsix_j.f54</i>
QX	The standard QX DSP file	The standard QX DSP file

See *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Use this function to initiate sending frequency shift key (FSK) data. When **parms** is set to NULL, the default parameter values are used. A typical buffer size is 512 bytes. The buffer size is limited to half the value of the maxbufsize field in the ADI_CONTEXT_INFO structure. The only baud rate supported is 1200.

Call **adiStopSendingFSK** to stop this function. ADIEVN_FSK_SEND_DONE is delivered when the send operation completes.

For more information, refer to *Sending and receiving FSK data* on page 69.

Example

```
#define MYSEND_STOPPED (-13)

int mySendFSK( CTAHD ctahd, void *buffer, unsigned bufsize )
{
    CTA_EVENT event;

    if( adiStartSendingFSK( ctahd, buffer, bufsize, NULL ) != SUCCESS )
        return MYFAILURE;

    do
    {
        myGetEvent( &event );      /* see ctaWaitEvent example */
    } while( event.id !=- ADIEVN_FSK_SEND_DONE );

    switch( event.value )
    {
        case CTA_REASON_FINISHED:
            return SUCCESS;

        case CTA_REASON_RELEASED:
            return MYDISCONNECT;

        case CTA_REASON_STOPPED:
            /* Send was stopped by another application thread */
            return MYSEND_STOPPED;

        default:
            if( CTA_IS_ERROR( event.value ) )
                return MYFAILURE;
    }
    return MYFAILURE;
}
```

adiStartSignalDetector

Starts detecting changes in incoming out-of-band signaling bits.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStartSignalDetector** (CTAHD *ctahd*, unsigned *initial*, unsigned *mask*, unsigned *timeon*, unsigned *timeoff*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>initial</i>	Mask indicating the expected incoming line state (refer to <i>mask</i> for possible values).
<i>mask</i>	Mask indicating the bits to monitor. For example, by setting this mask to ADI_A_BIT, all transitions of the A bit are reported and transitions of the other bits are ignored. The following constants are in <i>adidef.h</i> and can be combined using the OR operation to monitor any group of bits: ADI_A_BIT ADI_B_BIT ADI_C_BIT ADI_D_BIT
<i>timeon</i>	Deglitching (debounce) time (in milliseconds) for the ON state of the masked bits. The bit transition to HIGH is not reported unless it exceeds <i>timeon</i> .
<i>timeoff</i>	Deglitching (debounce) time (in milliseconds) for the OFF state of the masked bits. The bit transition to LOW is not reported unless it exceeds <i>timeoff</i> .

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state.
CTAERR_FUNCTION_ACTIVE	Function already active.
CTAERR_FUNCTION_NOT_AVAIL	Necessary .dsp file was not downloaded to the board.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not valid in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_SIGNALBIT_CHANGED	<p>After the detector is started, if a bit transition is detected, the ADI service generates ADIEVN_SIGNALBIT_CHANGED with the value field set to the bit change and the size field set to the current state of the signaling bits.</p> <p>The change is defined as follows:</p> <p>0xA1 A bit went HI</p> <p>0xB1 B bit went HI</p> <p>0xC1 C bit went HI</p> <p>0xD1 D bit went HI</p> <p>0xA0 A bit went LO</p> <p>0xB0 B bit went LO</p> <p>0xC0 C bit went LO</p> <p>0xD0 D bit went LO</p> <p>The current state is a mask of the bits ADI_A_BIT, ADI_B_BIT, ADI_C_BIT, and ADI_D_BIT. These messages are serialized with the transitions. You receive one event for each bit change.</p>
ADIEVN_SIGNAL_DETECT_DONE	<p>The value field can be set to any of the following:</p> <p>CTAERR_xxx or ADIERR_xxx Signal detector function failed.</p> <p>CTA_REASON_STOPPED Function stopped as a result of calling adiStopSignalDetector.</p>

Details

AG 2000, AG 2000C, and AG 2000-BRI boards require *signal.m54* to be loaded.

Use **adiStartSignalDetector** to enable detection of incoming out-of-band signaling bits. After this function is called, transitions of masked bits are reported as events, along with the current state of all bits.

If the line state does not match the value set in **initial**, an event is generated after qualification time, **timeon**, or **timeoff**.

Note: This function is incompatible with standard call control. Contexts running a standard protocol other than NOCC are usually excluded from using this function. Protocols usually use out-of-band signaling bits for call setup (detection of incoming calls) and call teardown (detection of hang-up). For more information about controlling calls under specific TCPs, refer to the *NMS CAS for Natural Call Control Developer's Manual*.

For more information, refer to *Performing low-level call control* on page 71.

See also**adiQuerySignalState****Example**

```

#define ALL_BITS (ADI_A_BIT|ADI_B_BIT|ADI_C_BIT|ADI_D_BIT)

int myMonitorSignal( CTAHD ctahd )
{
    CTA_EVENT event;
    /* start function to monitor all bit changes of 100 ms */
    if( adiStartSignalDetector( ctahd, 0, ALL_BITS, 100, 100 ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        const char *pc;

        myGetEvent( &event );                                /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_SIGNAL_DETECT_DONE:
                if( event.value == CTA_REASON_STOPPED )
                    return SUCCESS;
                else
                    return MYFAILURE;

            case ADIEVN_SIGNALBIT_CHANGED:
                switch( event.value )                        /* value contains the change */
                {                                           /* size contains current state */
                    case 0xA1: pc = "A ON"; break;
                    case 0xB1: pc = "B ON"; break;
                    case 0xC1: pc = "C ON"; break;
                    case 0xD1: pc = "D ON"; break;
                    case 0xA0: pc = "A OFF"; break;
                    case 0xB0: pc = "B OFF"; break;
                    case 0xC0: pc = "C OFF"; break;
                    case 0xD0: pc = "D OFF"; break;
                }
                printf( "MVIP bit change: %s\t signalling bits = 0x%x "
                        " (%c%c%c%c)\n",
                        pc, (event.value&0xf),
                        (event.size&0x8)?'A':'-', (event.size&0x4)?'B':'-',
                        (event.size&0x2)?'C':'-', (event.size&0x1)?'D':'-' );
                break;
            /* might include cases to handle disconnect event, DTMFs, etc. */
        }
    }
}

```

adiStartTimer

Starts (or restarts) a timer on the board.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartTimer** (CTAHD *ctahd*, unsigned *timeout*, unsigned *count*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>timeout</i>	Timeout value (in milliseconds).
<i>count</i>	Number of events.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_TIMER_DONE	After the timer completes (expires), the ADI service generates a DONE event with the value field set to CTA_REASON_FINISHED. If the board is in error, there is an error in the value field. The value is CTA_REASON_STOPPED if the timer is halted with adiStopTimer .
ADIEVN_TIMER_TICK	If <i>count</i> is greater than 1, the ADI service generates a tick event for the first (<i>count</i> -1) expirations. On the final expiration, ADIEVN_TIMER_DONE is generated.

Details

The ADI service supports one application timer per port. This on-board timer has 10 ms resolution. The timer can be used when the application is controlling the protocol from application space. The timer generates periodic events. Specify both the period (*timeout*) and number of events (*count*).

Stop the timer by calling **adiStopTimer**. Reset or restart the timer with another call to **adiStartTimer**. When the timer is restarted, previous timer definitions are discarded and the timer begins with the new parameters.

Note: Unlike most other ADI service asynchronous functions, the timer function is not stopped automatically when a call is released.

For more information, refer to *Using on-board timers* on page 72.

Example

```
int myTimer( CTAHD ctahd, unsigned ms )
{
    CTA_EVENT event;

    if( adiStartTimer( ctahd, ms, 1 /*count*/ ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );                /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_TIMER_DONE:
                if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;          /* API error */
                else
                    return SUCCESS;             /* stopped normally */
                break;

            /* might include cases to handle disconnect event, DTMFs, etc. */
        }
    }
}
```

adiStartToneDetector

Starts detecting a precise tone.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartToneDetector** (CTAHD **ctahd**, unsigned **toneid**, unsigned **freq1**, unsigned **bandw1**, unsigned **freq2**, unsigned **bandw2**, ADI_TONEDETECT_PARMS ***parms**)

Argument	Description
ctahd	Context handle returned by ctaCreateContext or ctaAttachContext .
toneid	ID or instance of the detector. The range is 1 through 6. If the current protocol is providing clear-down detection, toneid =1 is not available.
freq1	First (or only) frequency to detect (in Hz).
bandw1	Bandwidth of the first frequency (in Hz).
freq2	The second frequency (in Hz) if the tone contains two frequencies, otherwise zero.
bandw2	Bandwidth of the second frequency.
parms	<p>Pointer to tone detection parameters, as shown (NULL designates default values):</p> <pre>typedef struct { DWORD size; /* size of this structure */ INT32 qualampl; /* broadband qual level (in dBm) */ DWORD qualtime; /* qualification time (in ms) */ DWORD reflvel; /* qual thresh,output of filter (IDUs) */ DWORD reserved; /* reserved, must be 0 */ } ADI_TONEDETECT_PARMS;</pre> <p>Refer to ADI_TONEDETECT_PARMS on page 268 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not available in the current call state.
CTAERR_BAD_ARGUMENT	Function argument had an invalid value, or a required pointer argument was NULL.
CTAERR_FUNCTION_ACTIVE	Function already active.
CTAERR_FUNCTION_NOT_AVAIL	Necessary .dsp file was not downloaded to the board.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_TONE_1_BEGIN	Precise tone 1 detected on.
ADIEVN_CP_CED	Call progress analysis detected modem tone.
ADIEVN_CP_DIALTONE	Call progress analysis detected dial tone.
ADIEVN_CP_DONE	Call progress analysis complete.
ADIEVN_TONE_1_END	Precise tone 1 detected off.
ADIEVN_TONE_2_BEGIN	Precise tone 2 detected on.
ADIEVN_TONE_2_END	Precise tone 2 detected off.
ADIEVN_TONE_3_BEGIN	Precise tone 3 detected on.
ADIEVN_TONE_3_END	Precise tone 3 detected off.
ADIEVN_TONE_4_BEGIN	Precise tone 4 detected on.
ADIEVN_TONE_4_END	Precise tone 4 detected off.
ADIEVN_TONE_5_BEGIN	Precise tone 5 detected on.
ADIEVN_TONE_5_END	Precise tone 5 detected off.
ADIEVN_TONE_6_BEGIN	Precise tone 6 detected on.
ADIEVN_TONE_6_END	Precise tone 6 detected off.
ADIEVN_TONE_1_DETECT_DONE	Precise tone detector 1 terminated.
ADIEVN_TONE_2_DETECT_DONE	Precise tone detector 2 terminated.
ADIEVN_TONE_3_DETECT_DONE	Precise tone detector 3 terminated.
ADIEVN_TONE_4_DETECT_DONE	Precise tone detector 4 terminated.
ADIEVN_TONE_5_DETECT_DONE	Precise tone detector 5 terminated.
ADIEVN_TONE_6_DETECT_DONE	Precise tone detector 6 terminated.

Details

For AG and CG boards, **adiStartToneDetector** requires one of the following DSP files to be specified in the board keyword file, depending on the **toneid** specified:

CG boards	AG boards	Description
<i>dtmf.f54</i> <i>dtmfe.f54</i>	<i>ptf.m54</i>	For toneid 1.
<i>ptf.f54</i>	<i>ptf.m54</i>	For toneid 2 through 6. The CG board uses <i>dtmf/dtmfe</i> for toneid 2 when detecting a single tone.

For QX boards, this function is supported in the standard DSP file. Refer to the *QX 2000 Installation and Developer's Manual* for a table of MIPS usage for all functions.

Use this function to start detecting a precise tone, which consists of one or two frequencies. The precise tone is defined in terms of center frequency and bandwidth pairs, specified in Hz. Bandwidth is the total band around the center frequency (for example, +/- bandwidth/2).

After the detector is started, if the specified tone is detected, the ADI service generates a BEGIN event. If the tone stops, the ADI service generates an END event. The detector continues until it is stopped by **adiStopToneDetector**, which is followed by a DONE event.

You can change the minimum qualification time specified by *qualtime* in the *ADI_TONEDETECT_PARMS* structure.

To set a time limit on the detection, use **adiStartTimer** to generate a timeout event. Call **adiStopToneDetector** if a timeout occurred.

For more information, refer to *Detecting tones* on page 50.

Example

```

int myDetectDialtone( CTAHD ctahd )
{
    CTA_EVENT event;
    unsigned toneid = 2;
    unsigned frequency1 = 350;
    unsigned bandwidth1 = 50;
    unsigned frequency2 = 440;
    unsigned bandwidth2 = 50;

    if( adiStartToneDetector( ctahd, toneid, frequency1, bandwidth1,
                             frequency2, bandwidth2, NULL ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_TONE_2_BEGIN:
                adiStopToneDetector( ctahd, toneid );
                break;                    /* on TONE_DETECT_DONE, will return */
            case ADIEVN_TONE_2_DETECT_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;    /* API error */
                else
                    return SUCCESS;      /* stopped normally */
                break;

                /* might include cases to handle disconnect, DTMFs, etc. */
        }
    }
}

```

adiStartTones

Starts generating one or more tones.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStartTones** (CTAHD *ctahd*, unsigned *count*, ADI_TONE_PARMS **parms*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>count</i>	Number of entries in the <i>parms</i> array.
<i>parms</i>	<p>Pointer to an array of tones defined by the following structure (NULL designates default values):</p> <pre>typedef struct { DWORD size ; /* size of this structure */ DWORD freq1; /* first frequency (Hz) */ INT32 ampl1; /* level of first tone (dBm) */ DWORD freq2; /* second frequency (Hz) */ INT32 ampl2; /* level of second tone (dBm) */ DWORD ontime; /* on duration of DTMF tone (ms) */ DWORD offtime; /* off duration of DTMF tone (ms) */ INT32 iterations; /* times to repeat above; -1 = forever*/ } ADI_TONE_PARMS;</pre> <p>Refer to <i>ADI_TONE_PARMS</i> on page 267 for field descriptions.</p>

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not available in the current call state.
CTAERR_FUNCTION_ACTIVE	Function already active.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_OUTPUT_ACTIVE	Function failed because there is another active output function.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_TONES_DONE	<p>The value field contains any of the following reasons:</p> <p>CTAERR_XXX or ADIERR_XXX Tone generation failed.</p> <p>CTA_REASON_FINISHED Tones generated.</p> <p>CTA_REASON_STOPPED Tone generation stopped by adiStopTones.</p>

Details

The following DSP file must be loaded to the board before running **adiStartTones**:

For these boards...	Load this DSP file...
AG	<i>tone.m54</i>
CG	<i>tone.f54</i>
QX	The standard QX DSP file

See *DSP file summary* on page 269 for DSP file descriptions. Refer to the board installation and developer's manual for a table of MIPS usage for all functions.

Use this function to start generating a sequence of tones, each consisting of one or two frequencies and an iteration count. The DONE event is generated when the tone sequence completes.

Each tone within the sequence comprises an ontime and an offtime, as well as an iterations count, all of which are contained in the ADI_TONE_PARMS structure. The final iteration is complete when the offtime expires. To generate a tone continuously, set iterations to -1 and specify an offtime of 0 (zero).

Use **adiStopTones** to prematurely terminate tone generation.

For more information, refer to *Generating tones* on page 52.

See also

adiStartDTMF

Example

```
/* generates an Intralata Reorder SIT per BellCore */
int myPlaySITReorder( CTAHD ctahd )
{
    ADI_TONE_PARMs p[3] = {0};
    CTA_EVENT      event;
    int            tonecnt = 3;

    p[0].freq1 = 914; p[0].ampl1 = -24; p[0].ontime = 275; p[0].iterations = 1;
    p[1].freq1 = 1429; p[1].ampl1 = -24; p[1].ontime = 380; p[1].iterations = 1;
    p[2].freq1 = 1777; p[2].ampl1 = -24; p[2].ontime = 380; p[2].iterations = 1;

    if( adiStartTones( ctahd, tonecnt, p ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );          /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_TONES_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;    /* API error */
                else
                    return SUCCESS;      /* stopped normally */
                break;
        }
    }
}
```


adiStopCallProgress

Stops a call progress analysis operation.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStopCallProgress** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not available in the current call state.
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_CP_DONE	After the call progress analysis operation stops, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopCallProgress** to disable the call progress analysis operation started by **adiStartCallProgress**. After this function is called, call progress analysis events are not reported.

Call progress analysis cannot be restarted until the DONE event is received.

Example

```
int myStopCallProgress( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopCallProgress( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_CP_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;    /* API error */
                else
                    return SUCCESS;      /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */
        }
    }
}
```

adiStopCollection

Stops the asynchronous digit collection operation.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopCollection** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not available in the current call state.
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_COLLECTION_DONE	After digit collection terminates, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopCollection** to stop digit collection started with **adiCollectDigits**. When digit collection stops, ADIEVN_COLLECTION_DONE is generated. Any digits already collected are included in the event's buffer.

See also

adiFlushDigitQueue, **adiGetDigit**

adiStopDial

Stops the dial operation.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopDial** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_DIAL_DONE	After the dial operation stops, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopDial** to stop the dial function started by **adiStartDial**. You can restart the dial operation (and any other operation requiring voice output) after you receive the DONE event.

Example

```
int myStopDial( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopDial( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_DIAL_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;      /* API error */
                else
                    return SUCCESS;         /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */
        }
    }
}
```

adiStopDTMFDetector

Stops DTMF detection.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopDTMFDetector** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_DTMF_DETECT_DONE	After the detector stops, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopDTMFDetector** to disable detection of DTMFs. Detection is automatically enabled by the call control protocols upon transition to the ADI_CC_STATE_CONNECTED state. After this function is called, DTMF events are not reported. After the DONE event is received, restart the detector with **adiStartDTMFDetector**.

adiCollectDigits does not work if you disable DTMF detection. No digits are collected and no events are generated.

Example

```
int myStopDTMFDetector( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopDTMFDetector( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_DTMF_DETECT_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;     /* API error */
                else
                    return SUCCESS;       /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */

        }
    }
}
```

adiStopEnergyDetector

Stops the energy detector.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopEnergyDetector** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_ENERGY_DETECT_DONE	After the detector stops, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopEnergyDetector** to stop the low-level energy detector started by **adiStartEnergyDetector**. After this function is called, energy and silence transitions are not reported. You can restart the energy detector after you receive the DONE event.

For more information, refer to *Detecting energy* on page 66.

Example

```
int myStopEnergyDetector( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopEnergyDetector( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_ENERGY_DETECT_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;      /* API error */
                else
                    return SUCCESS;         /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */
        }
    }
}
```

adiStopMFDetector

Stops the MF detector.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStopMFDetector** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt was made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_MF_DETECT_DONE	After the detector stops, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopMFDetector** to disable detection of MFs. After this function is called, MF events are not reported. When the DONE event is received, restart the MF detector with **adiStartMFDetector**.

Example

```
int myStopMFDetector( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopMFDetector( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_MF_DETECT_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;      /* API error */
                else
                    return SUCCESS;        /* stopped normally */
                break;
        }
    }
}
```

adiStopPlaying

Stops the play operation.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopPlaying** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_PLAY_DONE	After playing stops, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopPlaying** to stop the play operation started by either **adiStartPlaying**, **adiPlayFromMemory**, or **adiPlayAsync**. When the DONE event is received, you can restart the play operation and any other operation requiring voice output.

For more information, refer to *Playing* on page 25.

adiStopProtocol

Stops the execution of a telephony protocol.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopProtocol** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_STOPPROTOCOL_DONE	When the protocol stops, the ADI service generates a DONE event with the value field set to CTA_REASON_FINISHED.

Details

Use **adiStopProtocol** to stop a protocol previously started with **adiStartProtocol**. You can stop the running protocol from any state. If the protocol is in the middle of a call, the call is aborted (abnormally), the outgoing line signaling is set to ADI_CC_STATE_IDLE, and the incoming signaling is ignored. All functions executing on the context that require being in the connected state are automatically terminated with CTA_REASON_RELEASED.

When the DONE event is returned, you can start a new protocol.

Example

```
int myStopProtocol( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopProtocol( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event ); /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_STOPPROTOCOL_DONE:
                if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE; /* API error */
                else
                    return SUCCESS; /* stopped normally */
                break;
        }
    }
}
```

adiStopReceivingFSK

Stops receiving frequency shift key (FSK) data.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStopReceivingFSK** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_FSK_RECEIVE_DONE	Generated by the ADI service when the FSK receive function terminates. The event value field contains: CTA_REASON_STOPPED Stopped by application request.

Details

Use **adiStopReceivingFSK** to stop the receipt of data initiated by **adiStartReceivingFSK**. For more information, refer to *Sending and receiving FSK data* on page 69.

adiStopRecording

Stops the recording operation.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopRecording** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_RECORD_DONE	After recording stops and the final buffer is presented, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED.

Details

Use **adiStopRecording** to stop the recording operation started by either **adiStartRecording**, **adiRecordToMemory**, or **adiRecordAsync**. You can restart recording when you receive the DONE event.

For more information, refer to *Recording* on page 20.

adiStopSendingFSK

Stops sending frequency shift key (FSK) data.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStopSendingFSK** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_FSK_SEND_DONE	Generated by the ADI service when the send operation terminates. The event value field contains CTA_REASON_STOPPED (stopped by application request).

Details

Use **adiStopSendingFSK** to abort the transmission of FSK data initiated by **adiStartSendingFSK**. For more information, refer to *Sending and receiving FSK data* on page 69.

adiStopSignalDetector

Stops the out-of-band signaling bit detector.

Supported board types

- QX
- AG
- CG

Prototype

DWORD **adiStopSignalDetector** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt was made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_SIGNAL_DETECT_DONE	After signal detection stops, the ADI service generates a DONE event with a value field of CTA_REASON_STOPPED.

Details

Use **adiStopSignalDetector** to disable detection of incoming out-of-band signaling bits. After calling this function, incoming out-of-band bit transitions are not reported.

This function is incompatible with standard call control. Contexts running a standard protocol other than NOCC are usually excluded from using this function. Protocols usually use out-of-band signaling bits for call setup (detection of incoming calls) and call teardown (detection of hang-up). For information about controlling calls under specific TCPs, refer to the *NMS CAS for Natural Call Control Developer's Manual*.

For more information, refer to *Performing low-level call control* on page 71.

See also**adiStartSignalDetector****Example**

```
int myStopSignalDetector( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopSignalDetector( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_SIGNAL_DETECT_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE; /* API error */
                else
                    return SUCCESS;      /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */
        }
    }
}
```

adiStopTimer

Aborts the timer operation.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopTimer** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_TIMER_DONE	After the timer operation stops, the ADI service generates a DONE event with a value field of CTA_REASON_STOPPED.

Details

Use **adiStopTimer** to abort the timer operation started by **adiStartTimer**. For more information, refer to *Using on-board timers* on page 72.

Example

```
int myStopTimer( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopTimer( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_TIMER_DONE:
                if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;           /* API error */
                else
                    return SUCCESS;           /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */

        }
    }
}
```

adiStopToneDetector

Stops a precise tone detector.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopToneDetector** (CTAHD *ctahd*, unsigned *toneid*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>toneid</i>	A specified instance of the detector to stop. Current range is 1 through 6, and corresponds to the <i>toneid</i> passed to adiStartToneDetector .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_TONE_1_DETECT_DONE	Precise tone detector 1 terminated.
ADIEVN_TONE_2_DETECT_DONE	Precise tone detector 2 terminated.
ADIEVN_TONE_3_DETECT_DONE	Precise tone detector 3 terminated.
ADIEVN_TONE_4_DETECT_DONE	Precise tone detector 4 terminated.
ADIEVN_TONE_5_DETECT_DONE	Precise tone detector 5 terminated.
ADIEVN_TONE_6_DETECT_DONE	Precise tone detector 6 terminated.

Details

Use **adiStopToneDetector** to deactivate a precise tone detector. When the detector stops, the ADI service generates a DONE event with the value field set to CTA_REASON_STOPPED. A specific DONE event is defined for each of six precise tone detectors.

You can restart the tone detector specified by the **toneid** when you receive the DONE event.

For more information, refer to *Detecting tones* on page 50.

Example

```
int myStopToneDetector( CTAHD ctahd )          /* stop detector #2 */
{
    CTA_EVENT event;

    if( adiStopToneDetector( ctahd, 2 ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );                /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_TONE_2_DETECT_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE;      /* API error */
                else
                    return SUCCESS;        /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */
        }
    }
}
```

adiStopTones

Stops generating tones.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiStopTones** (CTAHD *ctahd*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Attempt made to stop a function that was not running.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	Attempt made to stop a function that is already being stopped.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_TONES_DONE	When the tone generation function is stopped, the ADI service generates a DONE event with the value (reason) CTA_REASON_STOPPED.

Details

Use **adiStopTones** to terminate tone generation started by either **adiStartTones** or **adiStartDTMF**. You can restart tone generation, and any other functions requiring voice output, when you receive the DONE event.

For more information, refer to *Generating tones* on page 52.

Example

```
int myStopTones( CTAHD ctahd )
{
    CTA_EVENT event;

    if( adiStopTones( ctahd ) != SUCCESS )
        return MYFAILURE;

    while( 1 )
    {
        myGetEvent( &event );           /* see ctaWaitEvent example */

        switch( event.id )
        {
            case ADIEVN_TONES_DONE:
                if( event.value == CTA_REASON_RELEASED )
                    return MYDISCONNECT; /* call has been terminated */
                else if( CTA_IS_ERROR( event.value ) )
                    return MYFAILURE; /* API error */
                else
                    return SUCCESS;      /* stopped normally */
                break;

            /* might include cases to handle disconnect, DTMFs, etc. */
        }
    }
}
```

adiSubmitPlayBuffer

Submits a buffer of data for a play operation initiated by **adiPlayAsync**.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiSubmitPlayBuffer** (CTAHD *ctahd*, void **buffer*, unsigned *size*, unsigned *flags*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>buffer</i>	Pointer to buffer containing voice data to be played.
<i>size</i>	Size of <i>buffer</i> (bytes).
<i>flags</i>	Set to ADI_PLAY_LAST_BUFFER if the given buffer is the last in the message; otherwise set to 0.

Return values

Return value	Description
SUCCESS	
ADIERR_TOO_MANY_BUFFERS	Application is out of synchronization with the play operation. Submit buffers only when requested.
CTAERR_BAD_ARGUMENT	<i>buffer</i> is NULL.
CTAERR_BAD_SIZE	<i>size</i> is not a multiple of framesize for the encoding in adiPlayAsync .
CTAERR_FUNCTION_NOT_ACTIVE	Either voice is not playing or the play operation was not initiated by calling adiPlayAsync .
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	adiStopPlaying was already invoked or the ADI_PLAY_LAST_BUFFER flag was already set in a previous call to adiSubmitPlayBuffer or adiPlayAsync .
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_PLAY_BUFFER_REQ	Generated by the ADI service when a buffer with voice data is required.
ADIEVN_PLAY_DONE	<p>Generated by the ADI service when the play operation terminates with a reason (value field) of:</p> <p>CTAERR_XXX or ADIERR_XXX Play failed.</p> <p>CTA_REASON_DIGIT Aborted due to DTMF.</p> <p>CTA_REASON_FINISHED Application submitted buffer with ADI_PLAY_LAST_BUFFER set and the buffer was completely played.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p>

Details

Use **adiSubmitPlayBuffer** to asynchronously submit buffers, provided that the:

- Play operation was initiated by **adiPlayAsync**.
- Play operation is currently active.
- ADI_PLAY_LAST_BUFFER flag was not set for any buffer submission for the current playing instance.
- ADI service issued ADIEVN_PLAY_BUFFER_REQ to the application and the application did not subsequently submit a buffer (only one buffer can be submitted to a play operation at a time).

size can be arbitrarily large, but must be an integral multiple of the frame size for the selected encoding. For optimum performance, **size** must be the largest frame multiple that will fit in one board buffer. You can obtain this size by calling **adiGetEncodingInfo** (refer to the **maxbufsize** argument). If **size** is less than or equal to the board buffer size, you can re-use the buffer as soon as this function returns.

buffer can be set to NULL and **size** set to 0 (zero) only if the ADI_PLAY_LAST_BUFFER flag is set. In this case, the play operation terminates when the previously submitted buffer finishes.

See also

adiGetPlayStatus, **adiStopPlaying**

Example

Refer to the *playrec* demonstration program.

adiSubmitRecordBuffer

Supplies an empty buffer to an asynchronous record operation that was initiated using **adiRecordAsync**.

Supported board types

- QX
- AG
- CG
- PacketMedia HMP process

Prototype

DWORD **adiSubmitRecordBuffer** (CTAHD *ctahd*, void **buffer*, unsigned *size*)

Argument	Description
<i>ctahd</i>	Context handle returned by ctaCreateContext or ctaAttachContext .
<i>buffer</i>	Pointer into process memory where recorded voice data will be written.
<i>size</i>	Size of the buffer (bytes).

Return values

Return value	Description
SUCCESS	
ADIERR_TOO_MANY_BUFFERS	More than two buffers were submitted. A maximum of two buffers can be submitted to a record operation at any given time.
CTAERR_BAD_ARGUMENT	<i>buffer</i> is NULL.
CTAERR_BAD_SIZE	<i>size</i> is 0 (zero).
CTAERR_FUNCTION_NOT_ACTIVE	Either not recording or the recording operation was not initiated by calling adiRecordAsync .
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	adiStopRecording was already invoked.
CTAERR_INVALID_STATE	Function not available in the current port state.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
ADIEVN_RECORD_BUFFER_FULL	<p>Generated by the ADI service when a buffer is filled with recorded voice data. The event contains the following fields:</p> <p>buffer Pointer to a previously submitted user buffer.</p> <p>size Number of bytes recorded into buffer.</p> <p>value Flags; if the ADI_RECORD_BUFFER_REQ bit is set, more buffers are needed and the application must submit another empty buffer. If the ADI_RECORD_UNDERRUN bit is set, an underrun occurred. There was no new buffer to record information when this one completed.</p>
ADIEVN_RECORD_DONE	<p>Generated by the ADI service when the record operation terminates. The event size field contains the total number of bytes recorded during the function instance. The value field contains one of the following termination reasons or an error code:</p> <p>CTA_REASON_DIGIT Aborted due to DTMF.</p> <p>CTA_REASON_NO_VOICE Remote party never spoke.</p> <p>CTA_REASON_RELEASED Call terminated.</p> <p>CTA_REASON_STOPPED Stopped by application request.</p> <p>CTA_REASON_TIMEOUT Maximum record limit reached.</p> <p>CTA_REASON_VOICE_END Remote party stopped speaking.</p> <p>CTAERR_xxx or ADIERR_xxx Record failed.</p>

Details

Use **adiSubmitRecordBuffer** to asynchronously submit empty buffers to a record operation, provided that the:

- Record operation was initiated by **adiRecordAsync**.
- Record operation is active.
- Application does not already have two actively submitted buffers.

The ADI service truncates the **size** so that the effective size is a multiple of the encoding frame size selected in **adiRecordAsync**. If the effective size is zero, CTAERR_BAD_SIZE is returned.

See also

adiGetRecordStatus

Example

Refer to the *playrec* demonstration program.

6

Demonstration programs

Summary of the demonstration programs

Each demonstration program is shipped as an executable program with its source files and make files.

Note: The *incta* and *outcta* programs demonstrate placing inbound and outbound calls. Refer to the *Natural Access Developer's Reference Manual* for information about these demonstration programs.

The following demonstration programs are provided with Natural Access and the ADI service:

Program	Demonstrates...
<i>hostp2p</i>	A live voice connection between two ports using play and record functions.
<i>playrec</i>	Playing and recording using asynchronous and callback methods.
<i>threads</i>	A simple, multi-threaded answering machine.

Before you start the demonstration programs, ensure that

- Natural Access is properly installed.
- The board is executing.
- Switching is correctly configured.

Refer to the board installation and developer's manual for details on installing the board.

[ctademo.c and ctademo.h](#)

All of the demonstration programs use a common set of high-level functions contained in *ctademo.c* and *ctademo.h*. This demonstration code provides functions for initializing Natural Access, opening and closing ports, waiting for calls, placing calls, answering calls, performing record and playback operations, and collecting digits. Use these functions as base code for developing your applications with Natural Access. This library of functions is for demonstration only, and is subject to change without notice.

Host port to port connection: hostp2p

hostp2p demonstrates live voice connection between two ports using play and record functions. This program uses simultaneous play and record of small buffers to simulate a real-time voice connection between two voice calls. It uses the asynchronous play and record functions of the ADI service.

Usage

hostp2p options

where **options** are:

Option	Use this option to specify the...
-b n	Board number. Default = 0.
-B n	Second board number (if different).
-s [n:] m	First port DSP address. Default = 0:0. Specify only the timeslot.
-S [n:] m	Second port DSP address. Default = 0:1. Specify only the timeslot.
-p protocol	Protocol to run. Default = LPS0.
-P protocol	Second port protocol (if different).
-e n	Encoding type. Refer to <i>adidef.h</i> . Default = 10 (mu-law).
-f n	Buffer size (ms). Default = 60.
-d digits	Digits to dial on port 2 (if not NOCC).
-E len:tim	Echo cancellation length:adaptime. Default = 4:100.

Running hostp2p

This procedure assumes that you are testing on an AG 2000 board with loop start line interfaces connected to phone lines. *hostp2p* requires *rvoice.m54* and *echo.m54* for an AG 2000 board.

Ensure that the board keyword file is set to SwitchConnections = Yes or Clocking.HBus.ClockMode = STANDALONE for the board you are using. These settings ensure that the default DSP-to-line interface connections are set up by NMS OAM.

To run *hostp2p*:

Step	Action
1	<p>Start <i>hostp2p</i> by entering the following command at the prompt:</p> <pre>hostp2p -p lps0 -d digits</pre> <p><i>hostp2p</i> starts and the following information appears:</p> <pre>CTA host port to port voice Demo V 1.0 (Dec 8 1997) Port #1: Board 0 Stream 0 Slot 0 Protocol = lps0 Port #2: Board 0 Stream 0 Slot 1 Protocol = lps0 Encoding = 10 Buffer time = 60 msec Echocanceling length = 4 msec, adapt time= 100 msec Initializing and opening the CTA context... Daemon not running. Using process global default parms. Trace disabled. ----- Waiting for incoming call...</pre> <p><i>hostp2p</i> waits for an incoming call.</p>
2	<p>Place a call to the telephone line connected to port 0.</p> <p>The following information appears:</p> <pre>Incoming Call... Answering call... Call connected. ----- Placing a call to '5551212'...</pre> <p><i>hostp2p</i> places a call to the number you specified. When the called party answers, you have a connection.</p>

Play and record: playrec

playrec demonstrates voice play and record using asynchronous buffer submission and play and record callback routines. This demonstration operates in two phases: asynchronous voice play and record operations, and callback voice play and record operations.

If you do not specify a buffer size on the command line (-z), *playrec* retrieves the board physical buffer using **adiGetEncodingInfo**.

The demonstration is constructed so that the play and record functions are synchronous within the application. This is a single-port, single-threaded demonstration.

Note: **adiStartPlaying** and **adiStartRecording** (and consequently, this demonstration program) are not supported while Natural Access is running in client/server mode.

Usage

```
playrec [options]
```

where ***options*** are:

Option	Use this option to...
-?	Display command line options.
-h	Display command line options.
-b <i>n</i>	Specify the board number <i>n</i> . Default is 0.
-s <i>n:m</i>	Specify MVIP stream and timeslot. Default is 0:0.
-r <i>n</i>	Specify the maximum recording duration (in seconds).
-z <i>n</i>	Specify the application buffer size. Must be a multiple of NMS_24 frame size (62).

Featured functions

adiGetEncodingInfo, **adiPlayAsync**, **adiRecordAsync**, **adiStartPlaying**, **adiStartRecording**, **adiSubmitPlayBuffer**, **adiSubmitRecordBuffer**

Running playrec

The following procedure assumes that you are using an AG 2000 DID board with a 2500-type telephone connected to one of the lines.

To run *playrec*:

Step	Action
1	Navigate to the <code>\nms\ctaccess\demos\playrec</code> directory.
2	<p>Start <i>playrec</i> by entering the following command at the prompt:</p> <pre>playrec [-b <i>n</i> -s <i>n:m</i> -r <i>n</i> -z <i>n</i>]</pre> <p>Make sure that you specify the proper board and timeslot. The default value for both arguments is 0 (zero).</p> <p>You are prompted to record a brief message. The prompt is played using asynchronous buffer submission and ADIEVN_PLAY_BUFFER_REQ is displayed on your screen (assuming you did not specify an application buffer large enough to fit the whole prompt file).</p> <p>You can prematurely terminate the prompt by entering a touchtone.</p>
3	<p>At the record beep prompt, begin speaking.</p> <p>ADIEVN_RECORD_BUFFER_FULL displays on your screen for each buffer_size time period.</p>
4	<p>You can prematurely terminate the recording by entering a touchtone or by ceasing to speak.</p> <p>The recording you just made is played back. Again, the message ADIEVN_PLAY_BUFFER_REQ displays on your screen.</p>
5	Repeat steps 2 and 3. The process repeats using callback mode. Since Natural Access automatically invokes the callback routine, the displayed event messages are replaced with the corresponding callback events.

Note: This demonstration program enables you to experiment with buffer sizes. The encoding format for the files is ADI_ENCODE_NMS_24, which has a 62-byte frame size. Buffer sizes you specify with the `-z` option must therefore be multiples of 62.

Multi-threaded application: threads

threads demonstrates handling multiple ports using one thread per port. This demonstration application is a multi-threaded answering machine using *ctademo*. Each thread opens a port and repeatedly waits for calls on the port. Each time a call is received, it answers, plays the answering message, and hangs up.

Note: **adiStartPlaying** (and consequently, this demonstration program) is not supported when the application is running in client/server mode.

Usage

```
threads [options]
```

where **options** are:

Option	Use this option to specify the...
-b <i>n</i>	Board number <i>n</i> . Default is 0.
-s <i>n:m</i>	MVIP stream and timeslot for the first channel. Default is 0:0.
-n <i>nports</i>	Number of ports (and threads) to use. Default is 1.
-p <i>protocol</i>	Protocol to run. Default is lps0.
-f <i>filename</i>	Voice file to use for answering message. Default is <i>answer.vce</i> . Encoding is assumed to be ADI_ENCODE_NMS_24.

Featured functions

adiStartPlaying (in DemoPlayFile)

Running threads

Before running *threads*, verify that your system has the proper configuration. It must have the desired number of lines connected to loop start hybrids having the same MVIP stream and successive MVIP timeslots.

To run *threads*, type the following command at the prompt:

```
threads [-b n -s n:m -n nports -p protocol -f filename ]
```

Specify the MVIP stream, the lowest-numbered MVIP timeslot, and the number of timeslots to use.

The demonstration continues to answer all of the lines until you stop it by pressing **Ctrl+C**.

Note: Code in *threads* parses and documents the command line arguments and creates threads under various operating systems. **RunDemo** performs the call handling.

7

Errors

Alphabetical error summary

All ADI service functions return either SUCCESS, or an error code indicating that the function failed and the reason for the failure. ADI service error codes are defined in the *adidef.h* and *ctaerr.h* include files. The error codes are prefixed with ADIERR_ or CTAERR_.

For a complete list of CTAERR_ codes, refer to the *Natural Access Developer's Reference Manual*.

Asynchronous functions return SUCCESS if the input arguments and context state are valid for the given function and if the ADI service sent the command to the hardware providing the service. SUCCESS, in this case, means the function was initiated and a DONE event is generated for the function.

The following table alphabetically lists the ADI service errors:

Error name	Hex	Decimal	Description
ADIERR_CANNOT_CREATE_CHANNEL	0x00010006	65542	Unable to create a channel to the board due to either board or driver limitations.
ADIERR_INVALID_CALL_STATE	0x00010005	65541	Function not valid in the current call state. For example, many functions require the call to be in ADI_CC_STATE_CONNECTED.
ADIERR_INVALID_QUEUEID	0x00010002	65538	Bad queue or driver ID.
ADIERR_NO_DSP_PORT	0x00010007	65543	No input or output stream for requested function.
ADIERR_NO_DSP_RESOURCES	0x00010008	65544	Not enough free DSP resources to run the requested function.
ADIERR_NOT_ENOUGH_RESOURCES	0x0001000A	65546	Unable to obtain port resource from on-board resource management.
ADIERR_PLAYREC_ACCESS	0x00010001	65537	User callback routine for playing or recording returned a value other than SUCCESS or ADI_PLAY_LAST_BUFFER.
ADIERR_TOO_MANY_BUFFERS	0x00010004	65540	Attempted to submit a play or record buffer with two buffers pending.
ADIERR_UNKNOWN_BOARDTYPE	0x00010003	65539	Board type is unrecognized (adiGetBoardSlots).
CTAERR_BAD_ARGUMENT	0x00000007	7	Function argument had an invalid value, or a required pointer argument was NULL.

Error name	Hex	Decimal	Description
CTAERR_BAD_SIZE	0x0000000B	11	Size argument was too small to receive a data structure, or a play or record buffer was not a multiple of the framesize for the specified encoding.
CTAERR_BOARD_ERROR	0x00000003	3	Unexpected error occurred on the board. In most cases, ADIEVN_BOARD_ERROR contains the board error code.
CTAERR_DRIVER_OPEN_FAILED	0x00000014	20	Driver open failed because either the driver was not installed or the maximum number of opens was exceeded.
CTAERR_DRIVER_RECEIVE_FAILED	0x00000016	22	Error occurred retrieving an event from the driver, or no events were queued in the driver.
CTAERR_DRIVER_SEND_FAILED	0x00000017	23	Error returned by the driver in sending a message to the board. Occurs if the board was reset.
CTAERR_DRIVER_VERSION	0x00000015	21	Driver does not support the requested function.
CTAERR_FATAL	0x00000002	2	Internal error occurred in the Natural Access library.
CTAERR_FUNCTION_ACTIVE	0x0000000F	15	Attempt made to start an asynchronous function that is already started. Also returned if an attempt was made to get a digit or flush the digit queue while collecting digits.
CTAERR_FUNCTION_NOT_ACTIVE	0x0000000E	14	Attempt made to stop or modify a function that was not running. Also occurs when a function call is made to ADI call control when the NCC service is active.
CTAERR_FUNCTION_NOT_AVAIL	0x0000000D	13	<ul style="list-style-type: none"> Necessary .dsp or .tcp file was not downloaded to the board. Requested function required a service that is reserved for use by call control.
CTAERR_INVALID_BOARD	0x0000001A	26	Specified board number was not successfully configured.
CTAERR_INVALID_CTAHD	0x00000005	5	Context handle is invalid.
CTAERR_INVALID_SEQUENCE	0x00000013	19	<ul style="list-style-type: none"> Attempt was made to stop a function that is already being stopped. Play or record buffer was submitted when it was not expected.

Error name	Hex	Decimal	Description
CTAERR_INVALID_STATE	0x0000000C	12	Function is not valid in the current port state. For example, most functions require the port to be in ADI_STATE_STARTED, which is entered after starting a trunk protocol.
CTAERR_LOCK_TIMEOUT	0x0000001D	29	Thread lock timed out.
CTAERR_NOT_FOUND	0x0000000A	10	Specified parameter does not exist.
CTAERR_NOT_IMPLEMENTED	0x00000009	9	Function was not implemented.
CTAERR_OUTPUT_ACTIVE	0x0000001B	27	<ul style="list-style-type: none"> Open port failed because the stream and slot are already opened on another port. Output operation such as play failed because there is another active output function.
CTAERR_OUT_OF_MEMORY	0x00000006	6	Unable to allocate memory for driver or port context, for play or record buffers, or for temporary storage. When this error occurs in a DONE event, it can mean that there was insufficient memory on the board.
CTAERR_OUT_OF_RESOURCES	0x00000008	8	Unable to create shared resources.
CTAERR_SHAREMEM_ACCESS	0x00000010	16	Failed accessing shared memory.
CTAERR_SVR_COMM	0X00000041	65	Server communication error.

Numerical error summary

The following table numerically lists the ADI service errors:

Hex	Decimal	Error name
0x00000002	2	CTAERR_FATAL
0x00000003	3	CTAERR_BOARD_ERROR
0x00000005	5	CTAERR_INVALID_CTAHD
0x00000006	6	CTAERR_OUT_OF_MEMORY
0x00000007	7	CTAERR_BAD_ARGUMENT
0x00000008	8	CTAERR_OUT_OF_RESOURCES
0x00000009	9	CTAERR_NOT_IMPLEMENTED
0x0000000A	10	CTAERR_NOT_FOUND
0x0000000B	11	CTAERR_BAD_SIZE
0x0000000C	12	CTAERR_INVALID_STATE
0x0000000D	13	CTAERR_FUNCTION_NOT_AVAIL
0x0000000E	14	CTAERR_FUNCTION_NOT_ACTIVE
0x0000000F	15	CTAERR_FUNCTION_ACTIVE

Hex	Decimal	Error name
0x00000010	16	CTAERR_SHAREMEM_ACCESS
0x00000013	19	CTAERR_INVALID_SEQUENCE
0x00000014	20	CTAERR_DRIVER_OPEN_FAILED
0x00000015	21	CTAERR_DRIVER_VERSION
0x00000016	22	CTAERR_DRIVER_RECEIVE_FAILED
0x00000017	23	CTAERR_DRIVER_SEND_FAILED
0x0000001A	26	CTAERR_INVALID_BOARD
0x0000001B	27	CTAERR_OUTPUT_ACTIVE
0x0000001D	29	CTAERR_LOCK_TIMEOUT
0x00000041	65	CTAERR_SVR_COMM
0x00010001	65537	ADIERR_PLAYREC_ACCESS
0x00010002	65538	ADIERR_INVALID_QUEUEID
0x00010003	65539	ADIERR_UNKNOWN_BOARDTYPE
0x00010004	65540	ADIERR_TOO_MANY_BUFFERS
0x00010005	65541	ADIERR_INVALID_CALL_STATE
0x00010006	65542	ADIERR_CANNOT_CREATE_CHANNEL
0x00010007	65543	ADIERR_NO_DSP_PORT
0x00010008	65544	ADIERR_NO_DSP_RESOURCES
0x0001000A	65546	ADIERR_NOT_ENOUGH_RESOURCES

8 Events

Event data structure

The ADI service uses an asynchronous programming model to capitalize on the concurrent processing between board processors and the host CPU. In response to commands executed by the application, the ADI service generates events that indicate certain conditions or state changes. All events are represented as a `CTA_EVENT` C data structure, as shown:

```
typedef struct
{
    DWORD    id;           /* event id (ADIEVN_xxx in 'adidef.h') */
    CTAHD    ctahd;        /* context handle */
    DWORD    timestamp;    /* timestamp */
    DWORD    userid;       /* user id (defined by ctaCreateContext) */
    DWORD    size;         /* size of buffer if buffer != NULL */
    void     *buffer;      /* otherwise, may contain event */
    DWORD    value;        /* specific data */
    DWORD    objHd;        /* buffer pointer */
} CTA_EVENT;
```

This structure, returned by **ctaWaitEvent**, informs the application which event occurred on which context, and includes additional information specific to the event.

The `CTA_EVENT` structure contains the following fields:

Field	Description
id	ADI event code defined in the <i>adidef.h</i> header file. All ADI events are prefixed with <code>ADIEVN_</code> (for example, <code>ADIEVN_SOMETHING_HAPPENED</code>).
ctahd	Context handle (the same as the one returned from ctaCreateContext).
timestamp	Time when the event was created in milliseconds. Use ctaGetTimeStamp to interpret the value. The resolution for AG board events is 10 milliseconds. For CG board and PacketMedia HMP process events, the resolution is 1 millisecond.
userid	User-supplied value to ctaCreateContext . This field is unaltered by the ADI service and facilitates asynchronous programming. Its purpose is to correlate a port with an application object or context when events occur.
size	Size (bytes) of the area pointed to by buffer. If the buffer is NULL, this field can hold an event-specific value.
buffer	Pointer to data returned with the event. The field contains an application process address and the event's size field contains the actual size of the buffer.
value	Event-specific value.
objHd	Service object handle for the client side.

DONE events

A DONE event is a Natural Access event informing the application that an asynchronous function completed processing. DONE event codes are in the form ADI_**function**_DONE where **function** is the completed function (for example, PLAY, RECORD, COLLECTION). DONE events have no special physical or processing characteristics; they have the same physical structure and are retrieved identically as all other events.

An asynchronous function can return SUCCESS to the application when invoked and the function can later fail on the board. If the board detects an error when running a function, the ADI service delivers a DONE event to the application, and the event value field contains an error code.

Alphabetical event summary

The following table alphabetically lists the ADI service events:

Event name	Description
ADIEVN_BOARD_ERROR	Unexpected board error returned. This error can mean that the TCP initiated call clearing from an inappropriate state.
ADIEVN_BOARD_EVENT	Low-level board event returned.
ADIEVN_COLLECTION_DONE	Digit collection complete.
ADIEVN_CP_BUSYTONE	Call progress analysis detected busy.
ADIEVN_CP_CED	Call progress analysis detected modem tone.
ADIEVN_CP_DIALTONE	Call progress analysis detected dial tone.
ADIEVN_CP_DONE	Call progress analysis complete.
ADIEVN_CP_NOANSWER	Call progress analysis detected no answer (after ringing).
ADIEVN_CP_RINGTONE	Call progress analysis detected ring tone (remote alerting).
ADIEVN_CP_RINGQUIT	Call progress analysis detected ring, but it stopped.
ADIEVN_CP_REORDERTONE	Call progress analysis detected reorder tone (fast busy).
ADIEVN_CP_SIT	Call progress analysis detected SIT (special information tone).
ADIEVN_CP_VOICE	Call progress analysis detected voice.
ADIEVN_DIAL_DONE	Dial function complete.
ADIEVN_DIGIT_BEGIN	Raw DTMF digit detected on.
ADIEVN_DIGIT_END	Raw DTMF digit detected off.
ADIEVN_DTMF_DETECT_DONE	DTMF detector terminated.
ADIEVN_ECHOCANCEL_STATUS	Arrival of echo cancellation status information.
ADIEVN_ECHOCANCEL_TONE	Arrival of echo cancellation tone disabler information.
ADIEVN_ENERGY_DETECT_DONE	Energy detector terminated.
ADIEVN_ENERGY_DETECTED	Energy detector reporting energy.
ADIEVN_FSK_RECEIVE_DONE	FSK receive operation complete.
ADIEVN_FSK_SEND_DONE	FSK send operation complete.
ADIEVN_MF_DETECT_DONE	MF detector terminated.
ADIEVN_MF_DIGIT_BEGIN	MF digit detected on.
ADIEVN_MF_DIGIT_END	MF digit detected off.
ADIEVN_PLAY_BUFFER_REQ	Asynchronous request for a buffer to play.
ADIEVN_PLAY_DONE	Play operation complete.
ADIEVN_PULSE_DONE	Pulse function complete.
ADIEVN_QUERY_SIGNAL_DONE	Returned query of out-of-band signaling bits.

Event name	Description
ADIEVN_RECORD_BUFFER_FULL	Asynchronous buffer to write to disk.
ADIEVN_RECORD_DONE	Record operation complete.
ADIEVN_RECORD_EVENT	Information sent by the custom recording function. See adiCommandRecord for more detail.
ADIEVN_RECORD_STARTED	Record operation started.
ADIEVN_SIGNALBIT_CHANGED	Signal detector reporting a change.
ADIEVN_SIGNAL_DETECT_DONE	Signal detector terminated.
ADIEVN_SILENCE_DETECTED	Energy detector reporting silence.
ADIEVN_STARTPROTOCOL_DONE	Acknowledgment of start protocol.
ADIEVN_STOPPROTOCOL_DONE	Acknowledgment of stop protocol.
ADIEVN_TIMER_DONE	Timer function complete (expired).
ADIEVN_TIMER_TICK	Timer function reporting timer tick.
ADIEVN_TONE_1_BEGIN	Precise tone 1 detected on.
ADIEVN_TONE_1_DETECT_DONE	Precise tone detector 1 terminated.
ADIEVN_TONE_1_END	Precise tone 1 detected off.
ADIEVN_TONE_2_BEGIN	Precise tone 2 detected on.
ADIEVN_TONE_2_DETECT_DONE	Precise tone detector 2 terminated.
ADIEVN_TONE_2_END	Precise tone 2 detected off.
ADIEVN_TONE_3_BEGIN	Precise tone 3 detected on.
ADIEVN_TONE_3_DETECT_DONE	Precise tone detector 3 terminated.
ADIEVN_TONE_3_END	Precise tone 3 detected off.
ADIEVN_TONE_4_BEGIN	Precise tone 4 detected on.
ADIEVN_TONE_4_DETECT_DONE	Precise tone detector 4 terminated.
ADIEVN_TONE_4_END	Precise tone 4 detected off.
ADIEVN_TONE_5_BEGIN	Precise tone 5 detected on.
ADIEVN_TONE_5_DETECT_DONE	Precise tone detector 5 terminated.
ADIEVN_TONE_5_END	Precise tone 5 detected off.
ADIEVN_TONE_6_BEGIN	Precise tone 6 detected on.
ADIEVN_TONE_6_DETECT_DONE	Precise tone detector 6 terminated.
ADIEVN_TONE_6_END	Precise tone 6 detected off.
ADIEVN_TONES_DONE	Tone generation function complete.

Numerical event summary

The following table numerically lists the ADI service events:

Hex	Decimal	Event name
0x00012030	73776	ADIEVN_PLAY_BUFFER_REQ
0x00012031	73777	ADIEVN_RECORD_STARTED
0x00012032	73778	ADIEVN_RECORD_BUFFER_FULL
0x00012034	73780	ADIEVN_RECORD_EVENT
0x00012035	73781	ADIEVN_ECHOCANCEL_STATUS
0x00012036	73782	ADIEVN_ECHOCANCEL_TONE
0x00012040	73792	ADIEVN_DIGIT_BEGIN
0x00012041	73793	ADIEVN_DIGIT_END
0x00012048	73800	ADIEVN_MF_DIGIT_BEGIN
0x00012049	73801	ADIEVN_MF_DIGIT_END
0x00012050	73808	ADIEVN_CP_VOICE
0x00012051	73809	ADIEVN_CP_DIALTONE
0x00012052	73810	ADIEVN_CP_BUSYTONE
0x00012053	73811	ADIEVN_CP_REORDERTONE
0x00012054	73812	ADIEVN_CP_RINGTONE
0x00012055	73813	ADIEVN_CP_NOANSWER
0x00012056	73814	ADIEVN_CP_RINGQUIT
0x00012057	73815	ADIEVN_CP_SIT
0x00012059	73817	ADIEVN_CP_CED
0x00012070	73840	ADIEVN_TONE_1_BEGIN
0x00012071	73841	ADIEVN_TONE_1_END
0x00012072	73842	ADIEVN_TONE_2_BEGIN
0x00012073	73843	ADIEVN_TONE_2_END
0x00012074	73844	ADIEVN_TONE_3_BEGIN
0x00012075	73845	ADIEVN_TONE_3_END
0x00012076	73846	ADIEVN_TONE_4_BEGIN
0x00012077	73847	ADIEVN_TONE_4_END
0x00012078	73848	ADIEVN_TONE_5_BEGIN
0x00012079	73849	ADIEVN_TONE_5_END
0x0001207A	73850	ADIEVN_TONE_6_BEGIN

Hex	Decimal	Event name
0x0001207B	73851	ADIEVN_TONE_6_END
0x00012080	73856	ADIEVN_SILENCE_DETECTED
0x00012081	73857	ADIEVN_ENERGY_DETECTED
0x00012090	73872	ADIEVN_TIMER_TICK
0x000120A0	73888	ADIEVN_SIGNALBIT_CHANGED
0x000120EE	73966	ADIEVN_BOARD_EVENT
0x000120FF	73983	ADIEVN_BOARD_ERROR
0x00012111	74001	ADIEVN_STARTPROTOCOL_DONE
0x00012112	74002	ADIEVN_STOPPROTOCOL_DONE
0x00012130	74032	ADIEVN_PLAY_DONE
0x00012131	74033	ADIEVN_RECORD_DONE
0x00012140	74048	ADIEVN_COLLECTION_DONE
0x00012141	74049	ADIEVN_DTMF_DETECT_DONE
0x00012142	74050	ADIEVN_MF_DETECT_DONE
0x00012150	74064	ADIEVN_CP_DONE
0x00012170	74096	ADIEVN_TONE_1_DETECT_DONE
0x00012171	74097	ADIEVN_TONE_2_DETECT_DONE
0x00012172	74098	ADIEVN_TONE_3_DETECT_DONE
0x00012173	74099	ADIEVN_TONE_4_DETECT_DONE
0x00012174	74100	ADIEVN_TONE_5_DETECT_DONE
0x00012175	74101	ADIEVN_TONE_6_DETECT_DONE
0x00012180	74112	ADIEVN_ENERGY_DETECT_DONE
0x00012190	74128	ADIEVN_TIMER_DONE
0x000121A0	74144	ADIEVN_PULSE_DONE
0x000121A1	74145	ADIEVN_SIGNAL_DETECT_DONE
0x000121A2	74146	ADIEVN_QUERY_SIGNAL_DONE
0x000121B0	74160	ADIEVN_TONES_DONE
0x000121C0	74176	ADIEVN_DIAL_DONE
0x000121E0	74208	ADIEVN_FSK_RECEIVE_DONE
0x000121E1	74209	ADIEVN_FSK_SEND_DONE

Events ordered by category

This topic presents the ADI service events by category. The following fields are always assigned, regardless of the event:

- id
- ctahd
- timestamp
- userid

The remaining value, size, and buffer fields vary depending upon the event. If there is no relevant information for a field, it can be empty for the specific event. The buffer field is filled only if data is given to the application. Any events that yield data are noted. The value field can contain an error code if the operation is in error when started or if the function fails.

This topic presents:

- Administrative events
- Play and record events
- DTMF events
- MF events
- Call progress events
- Tone detector events
- Call control primitives
- Miscellaneous events

Administrative events

ID	Value field	Size field
ADIEVN_BOARD_ERROR	low word=parm0 high word=xx00	low word=parm1 high word=parm2
ADIEVN_BOARD_EVENT	low word=msgtyp high word=obj	low word=parm0 high word=parm1
ADIEVN_STARTPROTOCOL_DONE	ADI_REASON_xxx	
ADIEVN_STOPPROTOCOL_DONE	ADI_REASON_xxx	

Play and record events

ID	Value field	Size field
ADIEVN_PLAY_BUFFER_REQ	1=started 2=underrun	
ADIEVN_PLAY_DONE	ADI_REASON_XXX	Bytes played
ADIEVN_RECORD_BUFFER_FULL	1=buffer requested 2=underrun 3=both	Buffer size
ADIEVN_RECORD_DONE	ADI_REASON_XXX	Bytes recorded
ADIEVN_RECORD_EVENT		
ADIEVN_RECORD_STARTED	0 (zero) or ADI_RECORD_BUFFER_REQ	

Note: The CTA_EVENT.buffer field for ADIEVN_RECORD_BUFFER_FULL contains a data pointer.

DTMF events

ID	Value field	Size field
ADIEVN_COLLECTION_DONE	ADI_REASON_XXX	String length + 1
ADIEVN_DIGIT_BEGIN	0 through 9, A through D, * (asterisk), # (number sign)	
ADIEVN_DIGIT_END	0 through 9, A through D, * (asterisk), # (number sign)	
ADIEVN_DTMF_DETECT_DONE	ADI_REASON_XXX	

Note: The CTA_EVENT.buffer field for ADIEVN_COLLECTION_DONE contains a data pointer.

MF events

Size field is not applicable.

ID	Value field
ADIEVN_MF_DETECT_DONE	ADI_REASON_XXX
ADIEVN_MF_DIGIT_BEGIN	0 through 9, B through F
ADIEVN_MF_DIGIT_END	0 through 9, B through F

Note: See **adiStartMFDetector** for translation of MF events.

Call progress events

Size field is not applicable.

ID	Value field
ADIEVN_CP_BUSYTONE	
ADIEVN_CP_CED	
ADIEVN_CP_DIALTONE	
ADIEVN_CP_DONE	ADI_REASON_XXX
ADIEVN_CP_NOANSWER	
ADIEVN_CP_RINGQUIT	
ADIEVN_CP_RINGTONE	Number of rings
ADIEVN_CP_REORDERTONE	
ADIEVN_CP_SIT	
ADIEVN_CP_VOICE	ADI_CP_VOICE_XXX

Note: Number of rings is set to 1 on the first occurrence of the event. If the call progress stopmask is set to enable multiple ring events, this field contains a count of the number of rings.

Tone detector events

Size field is not applicable.

ID	Value field
ADIEVN_TONE_1_BEGIN	
ADIEVN_TONE_1_DETECT_DONE	ADI_REASON_xxx
ADIEVN_TONE_1_END	
ADIEVN_TONE_2_BEGIN	
ADIEVN_TONE_2_DETECT_DONE	ADI_REASON_xxx
ADIEVN_TONE_2_END	
ADIEVN_TONE_3_BEGIN	
ADIEVN_TONE_3_DETECT_DONE	ADI_REASON_xxx
ADIEVN_TONE_3_END	
ADIEVN_TONE_4_BEGIN	
ADIEVN_TONE_4_DETECT_DONE	ADI_REASON_xxx
ADIEVN_TONE_4_END	
ADIEVN_TONE_5_BEGIN	
ADIEVN_TONE_5_DETECT_DONE	ADI_REASON_xxx
ADIEVN_TONE_5_END	
ADIEVN_TONE_6_BEGIN	
ADIEVN_TONE_6_DETECT_DONE	ADI_REASON_xxx
ADIEVN_TONE_6_END	

Call control primitives

ID	Value field	Size field
ADIEVN_DIAL_DONE	ADI_REASON_xxx	
ADIEVN_FSK_RECEIVE_DONE	ADI_REASON_xxx	Buffer size
ADIEVN_FSK_SEND_DONE	ADI_REASON_xxx	
ADIEVN_PULSE_DONE	ADI_REASON_xxx	
ADIEVN_QUERY_SIGNAL_DONE	ADI_REASON_xxx	ADI_BIT_xxx
ADIEVN_SIGNALBIT_CHANGED	ADI_x_BIT_xxx	ADI_BIT_xxx
ADIEVN_SIGNAL_DETECT_DONE	ADI_REASON_xxx	

Note: The CTA_EVENT.buffer field for ADIEVN_FSK_RECEIVE_DONE contains a data pointer.

Miscellaneous events

ID	Value field	Size field
ADIEVN_ECHOCANCEL_STATUS		Size of ADI_ECHOCANCEL_STATUS_INFO
ADIEVN_ECHOCANCEL_TONE	low word=type of tone high word=frequency	
ADIEVN_ENERGY_DETECT_DONE	ADI_REASON_xxx	Event ID (if condition is FINISHED)
ADIEVN_ENERGY_DETECTED		
ADIEVN_SILENCE_DETECTED		
ADIEVN_TIMER_DONE	ADI_REASON_xxx	
ADIEVN_TIMER_TICK	Tick count	

Note: The CTA_EVENT.buffer field for ADIEVN_ECHOCANCEL_STATUS contains a data pointer.

9 Parameters

Overview of the ADI service parameters

The behavior of many ADI functions is controlled by multiple parameters. These parameters are grouped together into structures. Each parameter structure has a set of default values that is sufficient for many configurations. The parameters can be modified to:

- Enable or disable function features.
- Adapt the function for exceptional configurations.

For example, when recording voice data, the application programmer can alter the function's behavior by modifying any of the record parameters that specify

- Any subset of DTMF keys entered by the telephone caller that abort the function.
- Gain applied to the input signal.
- An initial timeout that defines the time in which the caller must start speaking before the operation terminates.
- The amount of silence after a caller has stopped speaking before the operation terminates.
- Record-synchronization prompt frequency, amplitude, and duration.
- Automatic gain control settings.

For QX boards, refer to the *QX 2000 Installation and Developer's Manual* for each category of structure and default parameters values.

For information about parameter management in Natural Access, refer to the *Natural Access Developer's Reference Manual*.

ADI_CALLPROG_PARMS

Dependent function: adiStartCallProgress

Field name	Type	Default	Units	Description
busycount	DWORD	4	count	Number of non-precise busy tones that must occur before busy or fast busy is reported. Valid range is 1 through 32767.
leakagetime	DWORD	8	ms	Do not modify.
maxbusy	DWORD	1500	ms	Threshold time defining the total time period (on time plus off time) for distinguishing between slow busy and ringing tone. Valid range is 0 through 32767.
maxreorder	DWORD	700	ms	Threshold time defining the total time period (on time plus off time) for distinguishing between fast busy (reorder) and slow busy. Valid range is 0 through 32767.
maxring	DWORD	3000	ms	Maximum duration of a tone to distinguish a ringing tone from a dial tone. Valid range is 0 through 32767.
maxringperiod	DWORD	8000	ms	Length of time of the last ringing tone plus the silence that follows, before call progress reports a ringing-ended event.
noiselevel	DWORD	0x14000	IDU	Do not modify.
precmask	DWORD	7	mask	<p>Mask to control which precise detectors to run. To form a value, use the OR operator with any of the following bit masks:</p> <ul style="list-style-type: none"> ADI_CPMSK_PRECISE_CED (0x0001): CED tone modem ADI_CPMSK_PRECISE_SIT (0x0002): SIT ADI_CPMSK_PRECISE_BUSY (0x0004): Busy and reorder tone (US) ADI_CPMSK_PRECISE_425 (0x0008) 425 Hz tone (busy and reorder tone, non-US) ADI_CPMSK_PRECISE_SITEXT (0x0010): SIT type reporting ADI_CPMSK_PRECISE_TDD (0x0020): TDD/TTY device ADI_CPMSK_PRECISE_NU (0x0040): Unassigned number <p>You can run only three of the four detectors concurrently. If you specify all four detectors, busy and reorder tones are determined by cadence alone, and only the SIT, CED, and TDD/TTY detectors are enabled. Busy and reorder tone (bit value 0x0004) and the 425 Hz tone selection (bit value 0x0008) are mutually exclusive. If you choose both, only the 425 Hz filter is in effect.</p>
precqualtime	DWORD	150	ms	Precise tone qualification time. All precise tones must be longer than this time to qualify.

Field name	Type	Default	Units	Description
qualtonetime1	DWORD	60	ms	Do not modify.
qualtonetime2	DWORD	80	ms	Do not modify.
qualvoicetime1	DWORD	60	ms	Do not modify.
qualvoicetime2	DWORD	60	ms	Do not modify.
ringcount	DWORD	7	count	Number of ring tones that must occur before NO_ANSWER is reported. Valid range is 1 through 32767.
silencelevel	INT32	-40	dBm	Maximum signal level that is considered to be silence. Valid AG board and CG board range is -46 through -34. Valid QX board range is -45 through 0.
silencetime	DWORD	1500	ms	Minimum length of a silent period after voice is detected before call progress reports a voice-ended event.
stopmask	DWORD	0	mask	Mask to control which events cause call progress to stop. A value can be formed by using the OR operation with any of the following values: <ul style="list-style-type: none"> • 0x0001 = ring tone • 0x0002 = ring end • 0x0004 = voice begin • 0x0008 = medium voice duration • 0x0010 = long voice duration • 0x0020 = extended voice duration • 0x0040 = voice end
timeout	DWORD	1000	ms	Maximum time that can elapse with no stimulus from the network before call progress stops with reason of timeout. Valid range is 1 through 65535. If the value is set to zero, the timer is disabled.
voicextended	DWORD	9000	ms	Minimum length of time voice must be detected before call progress reports an extended-voice event.
voicelong	DWORD	6000	ms	Minimum length of time voice must be detected before call progress reports a long-voice event.
voicemedium	DWORD	3000	ms	Minimum length of time voice must be detected before call progress reports a medium-voice event.
voicetonratio	DWORD	0x30000	IDU	Do not modify.

ADI_COLLECT_PARMS

Dependent function: adiCollectDigits

Field name	Type	Default	Units	Description
firsttimeout	DWORD	10000	ms	Maximum time to wait for the first digit. Use 0 to wait forever. Otherwise, the valid range is 1 through 2147483647.
intertimeout	DWORD	5000	ms	Maximum time to wait for any digit after the first digit. Use 0 to wait forever. Otherwise, the valid range is 1 through 2147483647.
terminators	DWORD	0x0C00	mask	Mask that specifies which digits cause collection to terminate. A value can be formed by using the OR operation with any of the values for the validDTMFs field. Use 0 to indicate no terminators.
validDTMFs	DWORD	0x07FF	mask	Mask that specifies the digits to collect; only specified digits are added to the collected digit string. Specify ADI_DIGIT_ANY to accept all digits. See <i>Valid DTMF values</i> on page 256 for information. Optionally, the value ADI_COLLECT_QUIETLY can be added to this parameter to suppress all but the final ADIEVN_DIGIT_BEGIN and ADIEVN_DIGIT_END events that are normally generated as each digit arrives.
waitendtone	DWORD	0	mask	Flag to indicate that collection ends at the trailing edge of the last digit. If 0, collection ends as soon as the final digit is detected. If 1, collection does not end until the end of the final digit.

Valid DTMF values

A value that combines all of the valid DTMF values can be formed by using ADI_DIGIT_ALL (0xFFFF). Values can also be formed by using the OR operation with any of the following values:

Digit	Name	Value
0	ADI_DIGIT_0	0x0001
1	ADI_DIGIT_1	0x0002
2	ADI_DIGIT_2	0x0004
3	ADI_DIGIT_3	0x0008
4	ADI_DIGIT_4	0x0010
5	ADI_DIGIT_5	0x0020
6	ADI_DIGIT_6	0x0040
7	ADI_DIGIT_7	0x0080
8	ADI_DIGIT_8	0x0100
9	ADI_DIGIT_9	0x0200
*	ADI_DIGIT_STAR	0x0400

Digit	Name	Value
#	ADI_DIGIT_POUND	0x0800
A	ADI_DIGIT_A	0x1000
B	ADI_DIGIT_B	0x2000
C	ADI_DIGIT_C	0x4000
D	ADI_DIGIT_D	0x8000

ADI_DIAL_PARMS

Dependent function: adiStartDial

Field name	Type	Default	Units	Description
breaktime	DWORD	60	ms	Break (on-hook) duration for dial pulses. Valid AG board and CG board range is 0 through 30000. Valid QX board range is 0 through 32767.
dialtonewait	DWORD	5000	ms	Maximum time to wait for dial tone (; character). Valid range is 0 through 65535.
dtmfamp1	INT32	-6	dBm	Amplitude of the low frequency component of the DTMF pair. Valid AG board, CG board, and PacketMedia HMP range is -54 through -3. Valid QX board range is -90 through 0.
dtmfamp2	INT32	-4	dBm	Amplitude of the high frequency component of the DTMF pair. Valid AG board, CG board, and PacketMedia HMP range is -54 through -3. Valid QX board range is -90 through 0.
dtmfofftime	DWORD	80	ms	Duration of the silence time between each digit. Valid AG board, CG board, and PacketMedia HMP range is 0 through 65534. Valid QX board range is 0 through 2047.
dtmfontime	DWORD	80	ms	Duration of each DTMF or MF digit. Valid AG board, CG board, and PacketMedia HMP range is 0 through 65534. Valid QX board range is 0 through 2047.
flashtime	DWORD	500	ms	Amount of time to assert the on-hook signaling pattern for a flash (! character). Valid AG board and CG board range is 0 through 65535. Valid QX board range is 0 through 32767.
interpulse	DWORD	700	ms	Inter-digit time for pulsed dialing. Valid AG board and CG board range is 0 through 30000. Valid QX board range is 0 through 32767.
longpause	DWORD	5000	ms	Amount of delay associated with the . (period) character. Valid range is 0 through 65535.
maketime	DWORD	40	ms	Make (off-hook) duration for dial pulses. Valid AG board and CG board range is 0 through 30000. Valid QX board range is 0 through 32767.

Field name	Type	Default	Units	Description
method	DWORD	0	mask	Type of signaling. <ul style="list-style-type: none"> 0=DTMF 1=Pulse (Not applicable for the PacketMedia HMP process.) 2=MF (US)
reserved	DWORD	0	internal	Do not modify.
shortpause	DWORD	2000	ms	Amount of delay associated with the , (comma) character. Valid range is 0 through 65535.
tonebandw1	DWORD	40	Hz	Bandwidth of the first frequency of the dial tone detector. Valid AG board, CG board, and PacketMedia HMP range is 20 through 800. Valid QX board range is 40 through 2000.
tonebandw2	DWORD	40	Hz	Bandwidth of the second frequency of the dial tone detector. Valid AG board, CG board, and PacketMedia HMP range is 20 through 800. Valid QX board range is 40 through 2000.
tonefreq1	DWORD	350	Hz	First (or only) dial tone frequency. Valid AG board, CG board, and PacketMedia HMP range is 330 through 3600. Valid QX board range is 1 through 4000.
tonefreq2	DWORD	440	Hz	Second dial tone frequency. Set this value to 0 (zero) to detect a single frequency. Valid AG board, CG board, and PacketMedia HMP range is 330 through 3600. Valid QX board range is 1 through 4000.
tonequalampl	INT32	-28	dBm	Minimum signal amplitude to qualify for dial tone detection. Valid AG board, CG board, and PacketMedia HMP range is -40 through 0. Valid QX board range is -48 through 0.
tonequaltime	DWORD	50	ms	Minimum duration of a qualified tone to be considered dial tone. Valid range is 0 through 32767.
tonereflevel	DWORD	0xB000	IDU	Do not modify.
tonetotaltime	DWORD	0	ms	Detects interrupted dial tones (stuttered dial tone) in certain countries. Defaults to 0 (zero), which indicates that dialing can proceed as soon as a dial tone is detected without waiting for stuttered dial tone to end. If set to a non-zero value, the value represents the total qualification time for dial tone, and the following occurs: <ul style="list-style-type: none"> Only precise dialtone detection is used. If dialtone disappears, requalify until dialtonewait expires. If dialtone lasts for totaltime, proceed with dialing.

ADI_DTMF_PARMS

Dependent function: adiStartDTMF

Field name	Type	Default	Units	Description
ampl1	INT32	-6	dBm	Amplitude of the low frequency component of the DTMF pair. Valid AG board, CG board, and PacketMedia HMP range is -54 through -3. Valid QX board range is -90 through 0.
ampl2	INT32	-4	dBm	Amplitude of the high frequency component of the DTMF pair. Valid AG board, CG board, and PacketMedia HMP range is -54 through -3. Valid QX board range is -90 through 0.
longpause	DWORD	5000	ms	Amount of delay associated with the . (period) character. Valid range is 0 through 65535.
offtime	DWORD	80	ms	Duration of the silence time between each DTMF digit. Valid AG board, CG board, and PacketMedia HMP range is 0 through 65534. Valid QX board range is 0 through 2047. Note: In some instances, the silence time increases by 20 ms.
ontime	DWORD	80	ms	Duration of each DTMF digit. Valid AG board, CG board, and PacketMedia HMP range is 0 through 65534. Valid QX board range is 0 through 2047.
shortpause	DWORD	2000	ms	Amount of delay associated with the , (comma) character. Valid range is 0 through 65535.

ADI_DTMFDETECT_PARMS

Dependent function: adiStartDTMFDetector

Field name	Type	Default	Units	Description
columnfour	DWORD	1	mask	Flag that indicates whether to detect the A, B, C, and D DTMF digits. Set this value to 1 to detect these digits, or 0 to ignore them.
offqualampl	INT32	-45	dBm	Minimum signal required to maintain recognition of a DTMF signal once recognition has started. Valid AG board, CG board, and PacketMedia HMP range is -51 through -15. Not used for QX boards.
offqualtime	DWORD	40	ms	Minimum duration of absence of a recognized DTMF signal before an end-of-digit event will be emitted. Valid AG board, CG board, and PacketMedia HMP range is 5 through 32767. Valid QX board range is 30 through 32766.
offthreshold	DWORD	0x92E0	IDU	Do not modify. Not used for QX 2000 boards.
onqualampl	INT32	-39	dBm	Minimum signal level recognized as a DTMF signal. Valid AG board, CG board, and PacketMedia HMP range is -51 through -15. Not used for QX boards.
onqualtime	DWORD	50	ms	Minimum duration of a recognized DTMF signal before a digit event is emitted. Valid AG board, CG board, and PacketMedia HMP range is 22 through 32767. Valid QX board range is 30 through 32766.
onthreshold	DWORD	0xCAB0	IDU	Do not modify. Not used for QX boards.

ADI_ENERGY_PARMS

Dependent function: adiStartEnergyDetector

Field name	Type	Default	Units	Description
autostop	DWORD	1	mask	Controls whether the energy detector continues running after the first event. Set this value to 1 to stop after the first event, or 0 to run continuously.
deglitch	DWORD	20	ms	Minimum time before a transition between silence and energy is recognized. Valid AG board, CG board, and PacketMedia HMP range is 0 through 32767. Not used for QX boards.
thresholdampl	INT32	-45	dBm	Minimum signal level that is considered to be energy. Anything below this level is considered to be silence. Valid AG board, CG board, and PacketMedia HMP range is -51 through -15. Valid QX board range is -45 through 0.

ADI_FSKRECEIVE_PARMS

Dependent function: adiStartReceivingFSK

Field name	Type	Default	Units	Description
baudrate	DWORD	1200	integer	Transmission baud rate. 1200 is the only valid value.
droptime	DWORD	5	ms	Minimum dropout to silence before a packet is considered terminated.
minlevel	INT32	-35	dBm	Required minimum receive level.
minmark	DWORD	10	ms	Minimum required initial mark and seizure time.

ADI_FSKSEND_PARMS

Dependent function: adiStartSendingFSK

Field name	Type	Default	Units	Description
baudrate	DWORD	1200	integer	Transmission baud rate. 1200 is the only valid value.
level	INT32	-14	dBm	Transmit output level.
marktime	DWORD	500	ms	Length of initial mark signal.
noseizureflag	DWORD	1	integer	Controls whether channel seizure is omitted. <ul style="list-style-type: none"> 0 = send channel seizure 1 = just send mark
seizetime	DWORD	1000	ms	Duration of channel seizure; ignored if noseizureflag = 1.

ADI_PLAY_PARMS

Dependent functions: adiStartPlaying, adiPlayFromMemory, adiPlayAsync

Field name	Type	Default	Units	Description
DTMFabort	DWORD	0xFFFF	mask	Mask that enables you to control which DTMFs abort play. See <i>Valid DTMF values</i> on page 256.
gain	INT32	0	dB	Gain applied to the encoded audio. Ignored for encoding types for which applied gain is not supported. Valid AG board, CG board, and PacketMedia HMP range is -54 through 24. Valid QX board range is -48 through 42.
maxspeed	DWORD	100	percent	Maximum speed that is used. Determines how much DSP processing power is allocated to the play function. The valid AG board and CG board range is 100 through 200. Ignored for encoding types for which speed modification is not supported. Not used for QX boards.
speed	DWORD	100	percent	Initial speedup or slowdown factor to apply to the encoded audio. The valid AG board and CG board range is 50 to maxspeed. Ignored for encoding types for which speed modification is not supported. Not used for QX boards.

ADI_RECORD_PARMS

Dependent functions: `adiStartRecording`, `adiRecordToMemory`, `adiRecordAsync`

Field name	Type	Default	Units	Description
AGCattacktime	DWORD	14	ms	Automatic gain control (AGC) attack time constant. This value affects how quickly the gain is reduced for loud signals. Valid AG board, CG board, and PacketMedia HMP range is 1 through 30000. Not used for QX boards.
AGCdecaytime	DWORD	304	ms	AGC decay time constant. This value affects how quickly the gain is increased for soft signals. Valid AG board, CG board, and PacketMedia HMP range is 1 through 30000. Not used for QX boards.
AGCenable	DWORD	0	integer	Flag to enable AGC. Set to 1 to enable AGC and 0 to disable it. Note: AGC must be disabled if you are using voice activity detection.
AGCsilenceampl	INT32	-49	dBm	Noise threshold for AGC. Gain adjustment is suspended for signals below this level. Valid AG board, CG board, and PacketMedia HMP range is -72 through 0. Not used for QX boards.
AGCtargetampl	INT32	-19	dBm	Target amplitude for AGC. Valid AG board, CG board, and PacketMedia HMP range is -72 through 0. Valid QX board range is -42 through 0.
beepampl	INT32	-20	dBm	Amplitude of the record beep tone. Valid AG board, CG board, and PacketMedia HMP range is -54 through 3. Valid QX board range is -90 through 0.
beepfreq	DWORD	1000	Hz	Frequency of the record beep tone. 0 disables the beep. Valid AG board, CG board, and PacketMedia HMP range is 200 through 3600. Valid QX board range is 0 through 4000.
beeptime	DWORD	200	ms	Duration of the record beep tone. 0 disables the beep. Valid AG board, CG board, and PacketMedia HMP range is 0 through 65535. Valid QX board range is 0 through 8000
DTMFabort	DWORD	0xFFFF	mask	Mask that enables you to control which DTMFs abort a record. See <i>Valid DTMF values</i> on page 256.
gain	INT32	0	dB	Gain applied to the signal before it is encoded. If automatic gain control (AGC) is enabled, this value is the initial gain when record is started. Valid AG board, CG board, and PacketMedia HMP range is -54 through 24. Valid QX board range is -42 through 48. Use zero to record with no gain. Values specified out of range are translated into one of the boundary values.

Field name	Type	Default	Units	Description
novoicetime	DWORD	5000	ms	Maximum length of silence at the beginning of a recording before record is stopped with a reason of CTA_REASON_NO_VOICE. Use 0 to disable this timer. Valid range is 0 through 65535. Bypass by passing a value of 0.
silenceampl	INT32	-45	dBm	Maximum signal level considered to be silence. Valid AG board, CG board, and PacketMedia HMP range is -51 through -15. Valid QX board range is -45 through 0. NMS recommends that you use the default values.
silencedeglitch	DWORD	100	ms	Maximum non-silent interval that is ignored by the silence detector. Any sounds that last longer than this value reset the silence detector. Valid AG board, CG board, and PacketMedia HMP range is 0 through 32767. NMS recommends that you use the default values. Not used for QX boards.
silencetime	DWORD	3000	ms	Maximum length of silence after audio energy is detected before record stops with a reason of CTA_REASON_VOICE_END. Use 0 to disable this timer. Valid range is 0 through 65535. Bypass by passing a value of 0.

ADI_START_PARMS

Dependent function: adiStartProtocol

Field name	Type	Default	Units	Description
callctl.blockmode	DWORD	0	mask	Not applicable.
callctl.debugmask	DWORD	0x0000	mask	Not applicable.
callctl.eventmask	DWORD	0x0000	mask	Not applicable.
callctl.mediamask	DWORD	0x001F	mask	Controls which functions are running or reserved when the call enters the connected (conversation) state. (The NOCC protocol enters this state immediately). Reserved indicates that the DSP MIPS are committed to the operation before the operation actually starts. The application must reserve DSP resources in advance by using this parameter for DTMF detection, silence detection, clear-down detection, and echo cancellation. See <i>callctl.mediamask valid values</i> on page 266.
clear-down.bandw1	DWORD	40	Hz	Not applicable.
clear-down.bandw2	DWORD	40	Hz	Not applicable.
clear-down.freq1	DWORD	350	Hz	Not applicable.
clear-down.freq2	DWORD	440	Hz	Not applicable.
clear-down.qualampl	INT32	-28	dBm	Not applicable.
clear-down.qualtime	DWORD	1000	ms	Not applicable.
clear-down.reflevel	DWORD	0xB000	IDU	Not applicable.
clear-down.reserved	DWORD	0	internal	Not applicable.
clear-down.tonecount	DWORD	0	integer	Not applicable.
clear-down.maxofftime	DWORD	0	ms	Not applicable.
clear-down.maxontime	DWORD	0	ms	Not applicable.
clear-down.minofftime	DWORD	0	ms	Not applicable.
clear-down.minontime	DWORD	0	ms	Not applicable.
dial.breaktime	DWORD	60	ms	Not applicable.
dial.dialtonewait	DWORD	5000	ms	Not applicable.
dial.dtmfampl1	INT32	-6	dBm	Not applicable.
dial.dtmfampl2	INT32	-4	dBm	Not applicable.
dial.dtmfofftime	DWORD	80	ms	Not applicable.
dial.dtmfontime	DWORD	80	ms	Not applicable.
dial.flashtime	DWORD	500	ms	Not applicable.
dial.interpulse	DWORD	700	ms	Not applicable.

Field name	Type	Default	Units	Description
dial.longpause	DWORD	5000	ms	Not applicable.
dial.maketime	DWORD	40	ms	Not applicable.
dial.method	DWORD	0	mask	Not applicable.
dial.reserved	DWORD	0	internal	Not applicable.
dial.shortpause	DWORD	2000	ms	Not applicable.
dial.tonebandw1	DWORD	40	Hz	Not applicable.
dial.tonebandw2	DWORD	40	Hz	Not applicable.
dial.tonefreq1	DWORD	350	Hz	Not applicable.
dial.tonefreq2	DWORD	440	Hz	Not applicable.
dial.tonequalampl	INT32	-28	dBm	Not applicable.
dial.tonequaltime	DWORD	50	ms	Not applicable.
dial.tonereflevel	DWORD	0xB000	IDU	Not applicable.
dtmfdet.columnfour	DWORD	1		Flag that indicates whether to detect the A, B, C, and D DTMF digits. Set this value to 1 to detect these digits, or 0 to ignore them.
dtmfdet.offqualampl	INT32	-45	dBm	Minimum signal required to maintain recognition of a DTMF signal once recognition starts. Valid AG board, CG board, and PacketMedia HMP range is -51 through 15. Not used for QX boards.
dtmfdet.offqualtime	DWORD	40	ms	Minimum duration of absence of a recognized DTMF signal before an end-of-digit event is emitted. Valid AG board, CG board, and PacketMedia HMP range is 5 through 32767. Valid QX board range is 0 through 32766.
dtmfdet.offthreshold	DWORD	0x92E0	IDU	Do not modify. Not used for QX 2000 boards.
dtmfdet.onqualampl	INT32	-39	dBm	Minimum signal level recognized as a DTMF signal. Valid AG board, CG board, and PacketMedia HMP range is -51 through -15. Not used for QX boards.
dtmfdet.onqualtime	DWORD	50	ms	Minimum duration of a recognized DTMF signal before a digit event is emitted. Valid AG board, CG board, and PacketMedia HMP range is 22 through 32767. Valid QX board range is 30 through 32766.
dtmfdet.onthreshold	DWORD	0xCAB0	IDU	Do not modify. Not used for QX 2000 boards.

Field name	Type	Default	Units	Description
echocancel.adapttime	DWORD	0	ms	Echo canceller adaptation time for echocancel.mode = 2. The valid AG board and CG board range is 100 through 1000. Lower values require more DSP processing power. Not used for QX boards.
echocancel.filterlength	DWORD	0	ms	Filter length of echo canceller for echocancel.mode = 2. Set this value to 0 to omit echo cancelling. Valid range is 0 through 20. Higher values require more DSP processing power.
echocancel.gain	INT32	0	dB	Amount of amplification applied to echo-cancelled output. Valid AG board and CG board range is -54 through 24. Done by the automatic gain control (AGC) module for QX boards.
echocancel.mode	DWORD	0	Bit field	Controls echo canceller operation. Set the mode to one of the following: <ul style="list-style-type: none"> • 0 = No echo cancellation. • 1 = Use internal defaults for filter length and adaptation time based on board type. • 2 = Use specified values. • 3 = Ignore specified filterlength and adapttime values for adiModifyEchoCanceller only. Additional values can be formed by using the OR operation. See <i>echocancel.mode valid values</i> on page 267.
echocancel.predelay	DWORD	0	ms	Output sample delay. Valid range is 0 through 9. For AG boards and CG boards, valid range is 0 through 20. For QX boards, the default value is 0.

callctl.mediamask valid values

A value can be formed by using the OR operation with any of the following values:

Value	Description
0x0001	Reserve DTMF detector. Not used for QX boards.
0x0002	Reserve silence detector. Not used for QX boards.
0x0004	Reserve cleardown detector. Not used for QX boards.
0x0008	Start DTMF detector.
0x0010	Start echo canceller.

echocancel.mode valid values

Value	Description
0x0004	Enable dynamic windowing. QX boards only.
0x0008	Enable echo suppressor.
0x0010	Do not reset echo canceller.
0x0020	Disable filter taps adaptation.
0x0040	Bypass echo cancellation. AG boards and CG boards only.
0x0080	Request status of echo canceller. AG boards and CG boards only.
0x0100	Enable auto-status event generation when status of echo canceller changes. AG boards and CG boards only.
0x0200	Enable comfort noise generation. AG boards and CG boards only.

ADI_TONE_PARMS**Dependent function: adiStartTones**

Field name	Type	Default	Units	Description
ampl1	INT32	-20	dBm	Amplitude of the first (or only) frequency component. Valid AG board, CG board, and PacketMedia HMP range is -54 through 3. Valid QX board range is -90 through 0.
ampl2	INT32	0	dBm	Amplitude of the second frequency component, if any. Valid AG board, CG board, and PacketMedia HMP range is -54 through 3. Valid QX board range is -90 through 0.
freq1	DWORD	1000	Hz	First (or only) frequency of the generated tone. Valid range is 200 through 3600.
freq2	DWORD	0	Hz	Second frequency of the generated tone, or 0 if the tone is a single frequency. If not 0, valid range is 200 through 3600.
iterations	INT32	1	integer	Number of times to repeat the alternating tone and silence period. A count of -1 means repeat forever. Otherwise the valid range is 1 through 32767.
offtime	DWORD	0	ms	Duration of silence between tones. Specify 0 for no off time. Valid AG board, CG board, and PacketMedia HMP range is 0 through 65535. Valid QX board range is 0 through 8000.
ontime	DWORD	200	ms	Duration of the tone. Valid AG board, CG board, and PacketMedia HMP range is 1 through 65535. Valid QX board range is 0 through 8000.

ADI_TONEDETECT_PARMS

Dependent function: adiStartToneDetector

Field name	Type	Default	Units	Description
qualampl	INT32	-28	dBm	Minimum signal level that is detected. Valid AG board, CG board, and PacketMedia HMP range is -40 through 0. Valid QX board range is -48 through 0.
qualtime	DWORD	500	ms	Minimum duration of a detected tone before an event is emitted. Also specifies the minimum duration of the absence of detected tone before a tone-ended event is emitted. The valid range is 0 through 32767.
reflevel	DWORD	0xB000	IDU	Do not modify.
reserved	DWORD	0	internal	Do not modify.

10 DSP files

DSP file summary

This topic lists the DSP files needed for particular ADI service functions. Specify the files to be loaded in the board keyword file. Use NMS OAM to load DSP files onto boards. For more information, refer to the board installation and developer's manual.

DSP files ending in *.dsp* have mu-law and A-law versions. The names shown here are for the mu-law version. The A-law files have *_a* appended to the file name. For example, the A-law version of *voice.dsp* is *voice_a.dsp*. Some DSP files have versions with *_j* appended to the file name. For example, the V.23 version of *adsir.dsp* is *adsir_j.dsp*.

Note: DSP files for CG 6000/C, CG 6100C, and CG 6500C boards use an *.f54* file extension. CG 6565/C and CG 6060/C boards use an *.f41* file extension.

AG boards	CG boards	Description
<i>adsir(_j).m54</i>	<i>adsir(_j).fxx</i>	Contains the caller ID function that decodes the modem burst occurring between the first and second rings on a loop start line. This file also contains the FSK data receiver. Use <i>adsir.fxx</i> if one of the loop start protocols is used and the parameter <i>adilps.cidsupport</i> is set to 1. Use this file for adiStartReceivingFSK . (_j) is the V.23 variant.
<i>adsix(_j).m54</i>	<i>adsix(_j).fxx</i>	Contains the FSK data transmitter. Use this file for adiStartSendingFSK . (_j) is the V.23 variant.
<i>callp.m54</i>	<i>callp.fxx</i>	Contains voice and tone detectors used for call progress detection and for general tone detection. Use <i>callp.fxx</i> if any outgoing or two-way trunk protocol is in use and for adiStartCallProgress .
<i>dtmf.m54</i>	<i>dtmf.fxx</i>	Contains the DTMF receiver and the energy/silence detector. Use <i>dtmf.fxx</i> for DTMF detection. The energy/silence detector is used by the record functions and by adiStartEnergyDetector .
<i>dtmfe.m54</i>	<i>dtmfe.fxx</i>	Is a variation of <i>dtmf.fxx</i> , optimized for use with the echo canceller (<i>echo.fxx</i>). <i>dtmfe.fxx</i> yields better talk-off resistance but requires the echo canceller to achieve the best cut through performance.
<i>echo.m54</i>	<i>echo.fxx</i>	Contains the echo cancellation function. The echo canceller removes reflected energy from the incoming signal, which improves DTMF detection and voice recognition while playing. Use <i>echo.fxx</i> if echo cancellation is enabled. See adiStartProtocol and the ADISTART_PARMs category. Note: Substitute <i>dtmfe.fxx</i> for <i>dtmf.fxx</i> when using the echo canceller.
<i>echo_v3.m54</i>	<i>echo_v3.fxx</i>	Provides higher performance and support for longer echo tails. Requires more resources than <i>echo.m54</i> and can decrease the number of ports.

AG boards	CG boards	Description
<i>echo_v4.m54</i>	<i>echo_v4.fxx</i>	Contains the echo cancellation functions available in <i>echo_v3.x54</i> , as well as comfort noise generation and tone disabling features.
None	<i>g723.fxx</i>	CG boards only. Contains ITU G.723.1 play and record functions for both 5.3 kbit/s and 6.3 kbit/s rates. The codec data is output as raw bytes of the encoded 30 ms frames.
<i>g726.m54</i>	<i>g726.fxx</i>	Contains ITU G.726 ADPCM play and record functions. G.726 is a standard for 32 kbit/s speech coding. Note: These functions require more DSP processing time than the functions in <i>voice.xxx</i> . You cannot run as many actively playing or recording contexts as you can with other speech encodings.
None	<i>g729.fxx</i>	CG boards only. Contains ITU G.729A play and record functions. The 8 kbit/s codec data is output as raw bytes of the encoded 10 ms frames.
<i>gsm_ms.m54</i>	<i>gsm_ms.fxx</i>	Contains play and record functions for MS-GSM speech encoding at 13 kbit/s.
<i>gsm_mspl.m54</i>	<i>gsm_mspl.fxx</i>	Similar in operation to <i>gsm_ms.m54</i> , except that maximum output of the play function is limited.
<i>ima.m54</i>	<i>ima.fxx</i>	Contains play and record functions for IMA ADPCM speech encoding, at 24 kbit/s or 32 kbit/s.
<i>mf.m54</i>	<i>mf.fxx</i>	Contains the multi-frequency receiver function. Required for any trunk protocol that uses MF signaling and also by adiStartMFDetector .
<i>oki.m54</i>	<i>oki.fxx</i>	Contains play and record functions for OKI ADPCM speech encoding, at 24 kbit/s or 32 kbit/s.
<i>ptf.m54</i>	<i>ptf.fxx</i>	Contains precise tone filters. On AG boards, loop start protocols use <i>ptf.xxx</i> for the clear-down detector. On AG and CG boards, use <i>ptf.xxx</i> for adiStartToneDetector and adiStartCallProgress .
<i>rvoice.m54</i>	<i>rvoice.fxx</i>	Contains PCM play and record functions. <i>rvoice.xxx</i> is required to play or record with an encoding of ADI_ENCODE_MULAW, ADI_ENCODE_ALAW, or ADI_ENCODE_PCM8M16.
<i>rvoice_vad.m54</i>	<i>rvoice_vad.fxx</i>	Contains PCM play and record functions. Record functions can enable voice activity detection. <i>rvoice_vad.xxx</i> is required to play or record with an encoding of ADI_ENCODE_MULAW, ADI_ENCODE_ALAW, or ADI_ENCODE_PCM8M16.
<i>signal.m54</i> (not required for AG 4000/C and AG 4040/C boards)	<i>qtsignal.f54</i> , for CG 6000/C boards <i>8tsignal.f54</i> , for CG 6100C and CG 6500C boards (not required for CG 6565/C and CG 6060/C boards)	Contains signaling, ring detector, and pulse functions. These are out-of-band functions that typically operate on the MVIP signaling stream. Required for any trunk protocol except NOCC. Also required for adiStartSignalDetector , adiQuerySignalState , and adiStartPulse .

AG boards	CG boards	Description
<i>tone.m54</i>	<i>tone.fxx</i>	Contains the tone generation function. Required for all trunk protocols except NOCC. Also required for adiStartTones , adiStartDTMF and adiStartDial , and for any record function to generate the beep.
<i>voice.m54</i>	<i>voice.fxx</i>	Contains NMS Communications ADPCM play and record functions. The compressed speech is in a framed format with 20 milliseconds of data per frame. Speech is compressed to 16, 24, or 32 kbit/s, or it is stored as uncompressed mu-law or A-law (64 kbit/s). This file is required to play or record with encoding values of ADI_ENCODE_NMS_16, ADI_ENCODE_NMS_24, ADI_ENCODE_NMS_32, or ADI_ENCODE_NMS_64.
None	None	Substitute these files for <i>voice.dsp</i> to apply speed up to NMS Communications ADPCM encoded speech.
<i>wave.m54</i>	<i>wave.fxx</i>	Contains play and record functions for PCM speech in formats commonly used in WAVE files, including 8 and 16 bit, 11 kilo-samples per second sampling.

Index

A

ADI_BOARD_INFO 97
ADI_CALLCTL_PARMS 174
ADI_CALLPROG_PARMS 154, 254
ADI_CLEARDOWN_PARMS 174
ADI_COLLECT_PARMS 82, 256
ADI_CONTEXT_INFO 106
ADI_DIAL_PARMS 158, 257
ADI_DTMF_PARMS 161, 259
ADI_DTMFDETECT_PARMS 163, 174, 260
ADI_ECHOCANCEL_PARMS 123, 174
ADI_ECHOCANCEL_STATUS_INFO 123
ADI_EEPROM_DATA 111
ADI_ENERGY_PARMS 165, 260
ADI_FSKRECEIVE_PARMS 179, 261
ADI_FSKSEND_PARMS 186, 261
ADI_PLAY_PARMS 261
 adiPlayAsync 132
 adiPlayFromMemory 136
 adiStartPlaying 170
ADI_PLAY_STATUS 115
ADI_RECORD_PARMS 262
 adiRecordAsync 141
 adiRecordToMemory 145
 adiStartRecording 182
ADI_RECORD_STATUS 117
ADI_START_PARMS 174, 264
ADI_TIMESLOT 100
ADI_TIMESLOT32 103
ADI_TONE_PARMS 198, 267
ADI_TONEDETECT_PARMS 194, 268
adiapi.lib 12
adiAssertSignal 80
adiCollectDigits 82
adiCommandEchoCanceller 85
adiCommandRecord 91
adidef.h 82, 85, 113, 189
ADIERR_XXX_XXX 237, 239
ADIEVN_CP_TDD 48, 154
ADIEVN_XXX_XXX 243, 245
adiFlushDigitQueue 95
adiGetBoardInfo 97
adiGetBoardSlots 100
adiGetBoardSlots32 103
adiGetContextInfo 106
adiGetDigit 109, 121
adiGetEEPromData 111
adiGetEncodingInfo 113
adiGetPlayStatus 115
adiGetRecordStatus 117
adiGetTimeStamp 119
adiInsertDigit 121
adimgr.lib 12
adiModifyEchoCanceller 123
adiModifyPlayGain 128
adiModifyPlaySpeed 130
adiPeekDigit 131
adiPlayAsync 132
adiPlayFromMemory 136
adiQuerySignalState 139
adiRecordAsync 141
adiRecordToMemory 145, 150
adiSetBoardClock 149
adiSetNativeInfo 150
adiStartCallProgress 154
adiStartDial 158
adiStartDTMF 161

- adiStartDTMFDetector 163
- adiStartEnergyDetector 165
- adiStartMFDetector 167
- adiStartPlaying 170
- adiStartProtocol 174
- adiStartPulse 177
- adiStartReceivingFSK 179
- adiStartRecording 182
- adiStartSendingFSK 186
- adiStartSignalDetector 189
- adiStartTimer 192
- adiStartToneDetector 194
- adiStartTones 198
- adiStopCallProgress 201
- adiStopCollection 203
- adiStopDial 204
- adiStopDTMFDetector 206
- adiStopEnergyDetector 208
- adiStopMFDetector 210
- adiStopPlaying 212
- adiStopProtocol 213
- adiStopReceivingFSK 215
- adiStopRecording 216
- adiStopSendingFSK 217
- adiStopSignalDetector 218
- adiStopTimer 220
- adiStopToneDetector 222
- adiStopTones 224
- adiSubmitPlayBuffer 226
- adiSubmitRecordBuffer 228
- AGC (automatic gain control) 25
- ASR (automatic speech recognition) 67
 - examples of echo cancellation 57
 - recommendations for controlling echo 64
- asynchronous transfers 18
 - playing voice data 27
 - terminating a record function 20

- automatic gain control (AGC) 25
- automatic speech recognition (ASR) 67
 - examples of echo cancellation 57
 - recommendations for controlling echo 64

B

- board configurations 78
 - echo cancellation 61
 - voice detection 68

C

- call progress 74
 - adiStartCallProgress 154
 - adiStopCallProgress 201
 - managing 43
- callback transfers 18
 - playing voice data 26
 - terminating a record function 20
- collecting digits 53
 - adiCollectDigits 82
 - adiFlushDigitQueue 95
 - adiGetDigit 109
 - adiInsertDigit 121
 - adiPeekDigit 131
 - adiStopCollection 203
 - functions, summary of 75
- controlling echoes 57
 - adiCommandEchoCanceller 85
 - adiModifyEchoCanceller 123
 - functions, summary of 75
- CTA_EVENT 241
- CTA_MVIP_ADDR 11
 - adiGetBoardInfo 97
 - adiGetBoardSlots 100
 - adiGetBoardSlots32 103
 - adiGetEEPromData 111
 - adiGetTimeStamp 119
 - adiSetBoardClock 149
 - adiSetNativeInfo 150

ctaCreateContext 10

ctaCreateQueue 10

ctademo.c 231

ctademo.h 231

CTAERR_XXX_XXX 237, 239

ctaGetTimeStamp 119

ctaInitialize 10

ctaOpenServices 11

ctaWaitEvent 241

 adiGetDigit 109

D

demonstration programs 231

 hostp2p 232

 playrec 234

 threads 236

detecting energy 66

 adiStartEnergyDetector 165

 adiStopEnergy Detector 208

detecting tones 50

 adiStartEnergyDetector 165

 adiStartToneDetector 194

 adiStopEnergyDetector 208

 adiStopToneDetector 222

 functions, summary of 74

 managing call progress 43

detecting voice activity 67, 91

DONE events 241

driver-only mode 12

DSP files 134, 269

DTMF 76

 collecting digits 53

 digit collection functions 75

 DTMFAbort mask 19

 echo cancellation examples 57

DTMFAbort 19

E

echo cancellation 57

encoding formats 13

 adiGetEncodingInfo 113

 adiPlayAsync 132

errors 237, 239

events 243, 245

 categories 247

 data structure 241

F

fax 64

FSK data 76

 adiStartReceivingFSK 179

 adiStartSendingFSK 186

 adiStopReceivingFSK 215

 adiStopSendingFSK 217

 sending and receiving 69

functions 79

 call progress 74

 configuration information 78

 digit collection 75

 DTMF and MF detection 76

 FSK data 76

 low-level call control 77

 on-board timers 77

 play 73

 record 73

 telephony protocol 73

 tone detection 74

 tone generation 75

G

gain 29

generating tones 52

 adiStartDTMF 161

 adiStartTones 198

 adiStopTones 224

 functions, summary of 75

H

hostp2p demonstration program 232

I

IP telephony gateways 57

L

libadiapi.so 12

low-level call control 77

- adiAssertSignal 80

- adiQuerySignalState 139

- adiStartDial 158

- adiStartPulse 177

- adiStartSignalDetector 189

- adiStopDial 204

- adiStopSignalDetector 218

- performing 71

M

MF 76

modems 64

N

native play and record 31

- adiSetNativeInfo 150

- buffer sizes 17

- DSP files 17

- encoding formats 13

Natural Call Control service 9

nccGetLineStatusInfo 106

nccStartProtocol 61

NOCC 139

O

OEM 111

on-board timers 77

- adiStartTimer 192

- adiStopTimer 220

- starting 72

- stopping 72

out-of-band signaling 71

- adiQuerySignalState 139

P

parameters 253

- ADI_CALLPROG_PARMS 254

- ADI_COLLECT_PARMS 256

- ADI_DIAL_PARMS 257

- ADI_DTMF_PARMS 259

- ADI_DTMFDETECT_PARMS 260

- ADI_ECHOCANCEL_STATUS_INFO
125

- ADI_ENERGY_PARMS 260

- ADI_FSKRECEIVE_PARMS 261

- ADI_FSKSEND_PARMS 261

- ADI_PLAY_PARMS 261

- ADI_RECORD_PARMS 262

- ADI_START_PARMS 264

- ADI_TONE_PARMS 267

- ADI_TONEDETECT_PARMS 268

playing voice data 25

- adiGetEncodingInfo 113

- adiGetPlayStatus 115

- adiModifyPlayGain 128

- adiModifyPlaySpeed 130

- adiPlayAsync 132

- adiPlayFromMemory 136

- adiSetNativeInfo 150

- adiStartPlaying 170

- adiStopPlaying 212

- adiSubmitPlayBuffer 226

- functions, summary of 73

playrec demonstration program 234

R

recording voice data 20

- adiGetEncodingInfo 113

- adiGetRecordStatus 117

- adiRecordAsync 141

- adiRecordToMemory 145

- adiSetNativeInfo 150

- adiStartRecording 182

- adiStopRecording 216
- adiSubmitRecordBuffer 228
- functions, summary of 73
- retrieving configuration information 78
 - adiGetBoardInfo 97
 - adiGetBoardSlots 100
 - adiGetBoardSlots32 103
 - adiGetContextInfo 106
 - adiGetEEPromData 111

S

- setting time 149
- simultaneous play and record 30
- single memory transaction 18
- speed 29
- system restrictions 30

T

- TCP 73
 - adiStartProtocol 174
 - adiStopProtocol 213
- TDD/TTY device 154, 254
- terminating play function 26
- test.vce 182
- threads demonstration program 236
- tone detection 50
- transferring data 18
- two-wire switching 64

U

- underruns 17

V

- voice activity detection 67, 91